A Course on Programming in FORTRAN IV

V. J. CALDERBANK

```
      10 CALL DYBYDX(X,Y0,DYDX0,NEQS)
         DO 4 I=1,NEQS
      4  Y1(I)=Y0(I)+STEP/3.0*DYDX0(I)
         CALL DYBYDX(X+STEP/3.0,Y1,DYDX1,NEQS)
         DO 5 I=1,NEQS
      5  Y1(I)=Y0(I)+STEP/6.0*DYDX0(I)+STEP/6.0*DYDX1(I)
         CALL DYBYDX(X+STEP/3.,Y1,DYDX1,NEQS)
         DO 6 I=1,NEQS
      6  Y1(I)=Y0(I)+STEP/8.0*DYDX0(I)+3.0*STEP/8.0*DYDX1(I
         CALL DYBYDX(X+STEP/2.0,Y1,DYDX2,NEQS)
         DO 7 I=1,NEQS
      7  Y1(I)=Y0(I)+STEP/2.0*DYDX0(I)-3.0*STEP/2.0*DYDX1(I.
        12.0*STEP*DYDX2(I)
         CALL DYBYDX(X+STEP,Y1,DYDX1,NEQS)
         DO 8 I=1,NEQS
      8  Y2(I)=Y0(I)+STEP/6.0*DYDX0(I)+2.0*STEP/3.0*DYDX2(I)
        1STEP/6.0*DYDX1(I)
         DOUBLE=.TRUE.
         DO 9 I=1,NEQS
C     COMPUTE ERROR ON EACH Y VALUE
         ERROR=ABS(0.2*(Y1(I)-Y2(I)))
         IF(ERROR.LE.ACC)GOTO 11
C     IF ACCURACY IS NOT REACHED THEN HALVE STEP LENGTH
         STEP=STEP/2.0
         NSREQD=2*NSREQD
         NSDONE=2*NSDONE
         GOTO 10
      11 IF(ERROR*64.0.GT.ACC)DOUBLE=.FALSE.
      9  CONTINUE
C     PREPARE FOR ANOTHER STEP
         X=X+STEP
         DO 12 I=1,NEQS
      12 Y0(I)=Y2(I)
         NSDONE=NSDONE+1
C     IF REQUIRED STEP COMPLETED THEN GOTO 13
         IF(NSDONE.GE.NSREQD)GOTO 13
         IF(.NOT.(DOUBLE.AND.NSDONE.EQ.(NSDONE/2)*2.AND.
        1NSREQD.GT.1))GOTO 10
```

Science Paperbacks

A course on programming in FORTRAN IV

A course on programming in FORTRAN IV

V. J. CALDERBANK

The Computing and Applied Mathematics Group
U.K.A.E.A., The Culham Laboratory
Culham, Abingdon, Berkshire

LONDON

CHAPMAN AND HALL

First published 1969 by
Chapman and Hall Ltd.,
11 New Fetter Lane, London EC4P 4EE
Reprinted 1970 and 1972
First published in Science Paperbacks 1969
Reprinted 1970, 1972, 1974, 1976 and 1979
Printed in Great Britain by
Fletcher & Son Ltd, Norwich

ISBN 0 412 20640 4

Distributed in the U.S.A. by
Halsted Press
a Division of John Wiley & Sons, Inc.
New York

Preface

A digital computer is able to produce a solution to a particular problem only if it is presented with a series of simple instructions that it is able to perform and which will, when obeyed in a specific order, produce the desired result. It is possible now for the computer user to write these instructions in a language particularly suited to his problem. This language can be translated by the computer into a logically equivalent machine language which the computer is able directly to interpret.

Many languages have been designed with the scientist, engineer and mathematician in mind, and it is the purpose of this book to describe the rules for programming a computer in one such scientifically oriented language – FORTRAN IV. The implementation of this language on the IBM 360 has been considered in particular, but where this differs from ASA FORTRAN IV it has been pointed out in the text.

This book is intended as a simple introduction for beginners, and it is felt that it should supplement the manuals that are available at most computer installations. Only an elementary knowledge of mathematics is assumed in the text, but the Appendices contain two complete FORTRAN programs which it is felt may be useful to more advanced programmers.

I wish to acknowledge the assistance of many of my friends and colleagues without whose comments and criticisms this book would have appeared in a much inferior form. In particular, I am indebted to my husband, Dr M. Calderbank (also of the Computing and Applied Mathematics Group, Culham), for checking all the examples and exercises on the IBM 360/65 computer and for his continued help and encouragement throughout the period this book was being written. I am also very grateful to Professor E. J. Burge, Head of the Physics Department of Chelsea College, University of London, for encouraging me to put to more permanent good my experience of advising FORTRAN users, and for suggesting many improvements to the text. Finally, for their detailed reading of the manuscript and for their invaluable criticisms, I wish to thank Mr R. J. Housden of the University of London Institute of Computer Science and Miss L. Diprose of the University of Sheffield.

V. J. CALDERBANK

CULHAM

Contents

CHAPTER ONE

Fundamentals of FORTRAN

1.1. Introduction

The purpose of this book is to teach the reader to program a computer in one particular language. It is not intended to discuss the way in which a modern computer is constructed or operated. However, a certain minimum knowledge of the structure of a computer system is required before any attempt can be made to program it, and therefore a brief introduction is given here.

Figure 1.1 shows a schematic diagram of the flow of information through a typical modern computer system. Information is presented to the computer via an *input device* such as a paper tape reader or card reader. This information is stored in the computer's main *core store* (or *central memory*), which is subdivided into a large number of distinct units of storage known as *words*. The *central processing unit* (CPU) controls the sequence of operations within the computer. It is able to access the information held in a word of the memory and initiate the appropriate action. It also controls the transfer of information between the central memory and the *arithmetic unit* which contains one or more *accumulators* or high-speed working stores in which arithmetic operations are performed. The final results of the computation are transferred from the central memory to an *output device* such as a lineprinter, card punch or paper tape punch.

The CPU is able to gain very fast access to any word of the computer's central memory. However, core store is very expensive, and consequently most modern computers use a mixture of core store and slower (but cheaper) *backing store*. Magnetic tapes, drums and discs are now the most commonly used backing store devices.

An automatic digital computer is able to produce a solution to a particular problem only if it is presented with a series of simple instructions that it is able to perform and which will, when obeyed in a specific order, produce the desired result. This sequence of instructions is referred to as a *program*, and it is the responsibility of the human programmer to present his problem to the computer in this rigid form. The circuitry of any particular computer is designed to obey only a limited number of basic instructions, such as addition, subtraction, multiplication, division and so on. Therefore a sophisticated mathematical computation can be performed on a computer only if it is capable of being broken down into a logical sequence of these basic

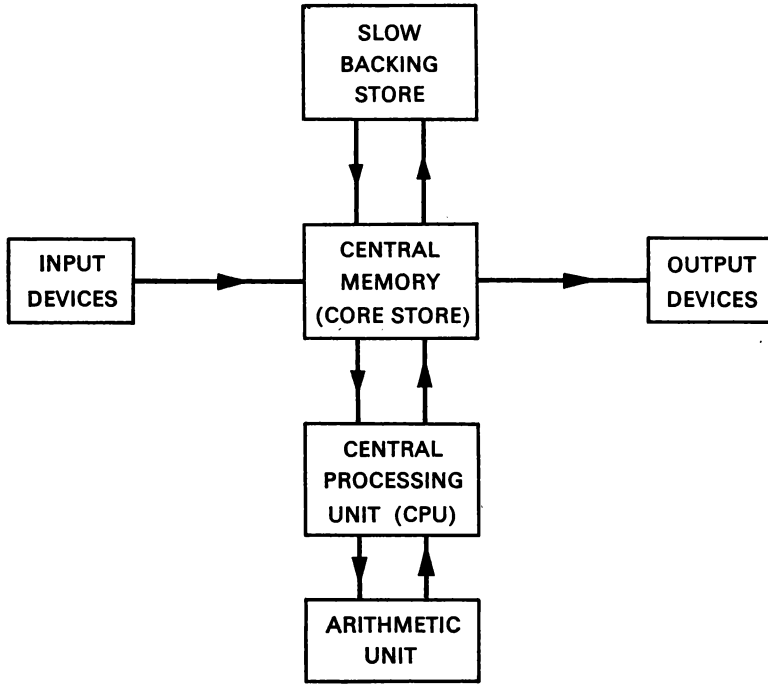


Figure 1.1 A schematic diagram showing the flow of information through a typical computer system

operations. Many numerical methods are concerned with the reduction of such problems as integration, minimization, the solution of equations, etc., to this simple numerical form. A mathematical process described in this way is generally known as an *algorithm*. It is the programmer's first task, therefore, to produce a suitable algorithm for his particular problem. A typical algorithm to find the sum of the squares of the integer numbers 1 to 100 might read as follows:

Set the value of SUM equal to zero.
Set the value of I equal to 1.

Start: If the value of I is greater than 100 end the process and print the value of SUM.

Otherwise square I and add the result to SUM.
Add 1 to I and return to the line labelled Start to continue the process.

Alternatively (or in addition), the algorithm can be produced in a diagrammatic form known as a *Flow Chart*. A flow chart for the above algorithm is shown in Figure 1.2. Note that certain conventions are

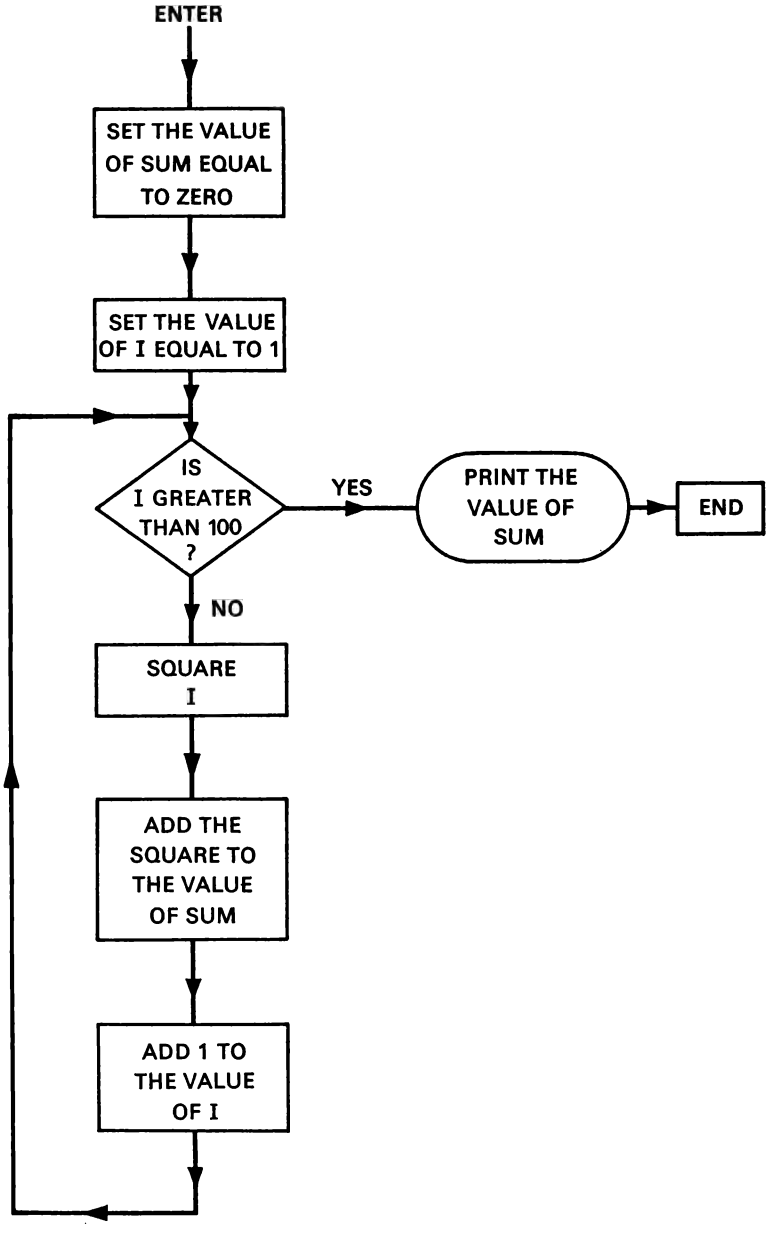


Figure 1.2 Flow chart of an algorithm to sum the squares of the integer numbers 1 to 100

generally followed in flow charts. All commands (e.g. add 1 to the value of I) are placed in rectangular boxes. Questions to be asked are placed in diamond-shaped boxes. These are generally referred to as *decision boxes*, since a decision has to be made at this point as to whether the answer to the question is *yes* or *no*. Thus there are always at least two ways out of any decision box, one to a series of instructions to be performed if the answer is *yes* and one to another series if the answer is *no*. Input or output instructions (e.g. print the value of SUM) are usually placed in rounded boxes. Arrows indicate the flow of the logic from box to box.

So far the programmer has been only concerned with crystallizing his ideas and formulating his problem in a suitable way. He must now produce the program – that is his sequence of instructions must be written in a language which the computer can understand. Any computer is constructed in such a way that it can fundamentally understand only one language – the *machine language* of that computer. Since programming in this basic machine language is a tedious and error-prone process, a multitude of so-called *high-level languages* have been designed to ease the programmer's task. A program written in one of these languages cannot be understood directly by the computer, and therefore it must be translated from the high-level language into the machine language of the computer. This task is performed by a program resident in the computer and known as the *compiler* (or *translator*). The compiler takes a *source* program in the high-level language and produces a logically equivalent *object* program in the machine code. This process is known as compiling the program and obeying the resulting sequence of machine code instructions is known as *executing* (or running) the program. From this it can be appreciated that any program written in a high-level language cannot be run on a computer which does not have a compiler for that particular language or version of that language.

It is the purpose of this book to describe the grammatical rules for writing programs in one particular scientifically oriented high-level language – FORTRAN. The FORTRAN project was started in the summer of 1954, and its aim was to produce a high-level language and compiler specifically for the IBM 704 computer. The resulting language was known as FORTRAN II, and since that time it has been developing steadily and has been implemented on many computers. An extended version of the language – FORTRAN IV – is now in widespread use. Since the FORTRAN IV language developed over a period of years, it was found that, although the language was in widespread use, there was no universally recognized definition of it and the version of FORTRAN IV recognized by a compiler at one computer installation could be quite different from that recognized by another compiler somewhere else. To remedy this, the American Standards Association produced a

definition of the FORTRAN IV language (ASA FORTRAN) which is now being adopted as a subset of most FORTRAN IV compilers. It is intended to describe here the FORTRAN IV language as implemented on the IBM 360 computer (see Form C28-6515-5 of the IBM Systems Reference Library – *The IBM System/360 FORTRAN IV Language*). Some of the more advanced facilities provided in this implementation will not be included, since it is felt that these will probably not be widespread. No attempt will be made to discuss in full how this description differs from other dialects of FORTRAN IV. However, the reader's attention will be drawn to certain areas of the language where differences are likely to arise. In particular, the differences between ASA FORTRAN IV and FORTRAN IV as discussed in this book will be pointed out. For convenience the language will be referred to throughout this book as FORTRAN.

1.2 The FORTRAN alphabet

Figure 1.3 shows the appearance of a typical FORTRAN program to find the roots of a quadratic equation of the form

$$ax^2 + bx + c = 0.$$

These are given by the expression

$$x = \frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a}$$

Since FORTRAN is a language, like any other language, it must have an alphabet in which it can be written down – or more important in this case, an alphabet in which it can be typed. FORTRAN programs are presented to the computer in the form of holes on punched cards or tape. Thus the FORTRAN alphabet must be selected from the available character set on a card punch and paper tape punch. The alphabet consists of:

(1) The decimal digits 0 to 9.
(2) The 26 upper case letters A to Z.
(3) The arithmetic operators +−*/** where / denotes divide, * multiply and ** is taken as a single symbol denoting exponentiation.
(4) Round brackets (and).
(5) The punctuation marks . and ,.
(6) Space and equal sign =.

It can be seen from Figure 1.3 that a FORTRAN program consists of a series of statements or lines, and the alphabet in which these can be physically represented has been described. Let us now study some of

the basic elements which are used to construct these statements – constants and variables. After learning how to combine these into statements we shall be in a position to write a simple program and to understand the structure of the program shown in Figure 1.3.

```
 1  C TO FIND THE ROOTS OF A QUADRATIC EQUATION
 2        READ(5,1)A,B,C
 3    1   FORMAT(3F10.5)
 4  C THE NEXT STATEMENT PROTECTS AGAINST A
 5  C POSSIBLE DIVISION BY ZERO LATER.
 6        IF(A.EQ.0)STOP
 7        B2M4AC = B**2-4.*A*C
 8  C THE NEXT STATEMENT CHECKS FOR IMAGINARY ROOTS.
 9        IF(B2M4AC.LT.0)GOTO 2
10        B2M4AC = SQRT(B2M4AC)
11        A2 = 2.*A
12        SOLN1 = (- B + B2M4AC)/A2
13        SOLN2 = (- B - B2M4AC)/A2
14        WRITE(6,3)SOLN1,SOLN2
15    3   FORMAT(14H0THE ROOTS ARE ,2F10.5)
16        STOP
17  C IF ROOTS ARE COMPLEX THEN OUTPUT WARNING CAPTION.
18    2   WRITE(6,5)
19    5   FORMAT(23H0THE ROOTS ARE COMPLEX.)
20        STOP
21        END
```

Figure 1.3 A FORTRAN program to find the roots of a quadratic equation

1.3 Numerical constants

Numerical constants in a FORTRAN program can be of three types – either integer, real or complex. An integer constant is a string of decimal digits written without a decimal point and optionally preceded by a + or − sign. Unsigned constants are taken to be positive. The magnitude must not be greater than an allowed maximum which varies with the machine.

The following are all examples of valid integer constants:

237 −82 +136 0028 0

but these are not:

23.0 (decimal point not allowed)
−3,500 (comma not allowed).

Real constants are distinguished from integer constants by the presence of a decimal point which may be at the beginning or end of a number or between any two digits. They are optionally preceded by a + or − sign. Real constants may be represented either in this *fixed*

point form or in *floating point* form, i.e. a real constant as just defined followed by an *exponent*. The exponent (written as the letter E followed by a one- or two-digit integer constant) represents the power of 10 by which the number is to be multiplied. Note that although the only restriction the compiler places on the exponent is that it must be a two-digit number, this does not mean that it can be as large as 99. There is a further restriction placed on the size of the exponent by the *word length* (i.e. the number of binary digits or *bits* in one word) of the machine. On the IBM 360 the largest exponent allowed is approximately +75. Anything larger than this will cause a fault of 'exponent overflow'.

Valid real constants are:

2.163 +0. −59.E−2 −.59 +28. −101.35
.002163E3 +0.00028E+5 −10135.0E−2

The following are not valid:

2,163.2 (comma not allowed)
5E20 (no decimal point)
59 (no decimal point)
22.0E150 (more than two digits in the exponent)
.59E2.5 (exponent not an integer)
E10 (exponent alone not allowed; this must be written 1.E10).

Real constants written in a double precision form and complex constants are discussed in a later chapter.

1.4 Variables

In mathematical equations (such as $y = 2x^2 + 4x + 1$) one not only uses constants (2, 4 and 1) but also variable quantities (x and y) which can take on a number of different values. It is obviously necessary, therefore, that any mathematical programming language should provide a similar facility. FORTRAN allows programmers to invent names for variables which can be of two types – real or integer. These names refer to locations in the computer (i.e. words in the memory) the contents of which may be real or integer numbers and may be changed at different points in the program. The programmer is at liberty to choose whatever names he likes provided the following rule is obeyed:

A variable name may consist of one to six letters or digits beginning with a letter. Note that characters other than letters or digits are not allowed. If the first letter is I,J,K,L,M,N, then the variable is an

integer variable. Names beginning with any other letter refer to real quantities.

It is always important to keep in mind the difference between real and integer quantities and to think of integer and real arithmetic as two separate processes. We shall see later (Chapter 2) that mixing integer and real variables in expressions is in general not allowed in FORTRAN, although it is allowed in IBM 360 FORTRAN IV and in many other recent compilers.

The values associated with real variables are stored in the machine in the same way as floating point constants, i.e. as a fractional part with an exponent.

Examples of correct variable names in FORTRAN are:

B24AC DERIV ALPHA XEQ16 FNX1 FNX2 XFN

for real variables and

INDEX NOVAR M4 K6X29Z

for integer variables. Incorrect variable names are:

2X139 (does not begin with a letter)

PRODUCT (contains more than six characters)

MK3/1 (contains a character other than a letter or a digit).

Note that every character and its position is significant. Thus FNX1, FNX2 and XFN are all quite distinct variables referring to different locations in the computer.

It is good practice when writing programs to select names of variables which describe the quantities they represent. Thus we could write down the equation for electrical power by letting the variable A2XZ represent watts, CPX represent volts and Q amps, i.e.

```
A2XZ=CPX*Q
```

However, the program would be far more readable if we chose WATTS, VOLTS and AMPS as the names of the variables and wrote

```
WATTS=VOLTS*AMPS
```

Although the programmer is given complete freedom in inventing the names of variables, he must take care not to use words which are reserved by the system for some particular purpose, e.g. system functions such as SQRT or SIN. This will be discussed more fully later.

EXERCISES 1

(1) Which of the following are correct representations of:
(*a*) integer constants; (*b*) real constants; and (*c*) neither?
Give reasons for (*c*).

5.2E100 2225 .137 2E—7 —25 1.562
E—10 —97612 101.2E+2. —.137 10000
0 +0162.2E—02 005326 +258. 3.6E—22 2,360

(2) Which of the following are correct names for: (*a*) integer variables;
(*b*) real variables; and (*c*) neither? Give reasons for (*c*).

PI 2X13 NUMBER M63—2 BSQUARED AMPS
FRED I123 ZETA X+Y ITER K(2) Q
IP COUNTS INC* NUMBER1 L3.6 POWER
L1369P JIM'S

CHAPTER TWO

Construction of a simple FORTRAN program

2.1 Arithmetic expressions

The simplest form of arithmetic expression in FORTRAN is a single operand (i.e. a real or integer variable or constant). A single operand can also be combined with other operands by arithmetic operators to form more complicated expressions. FORTRAN provides five arithmetic operators represented by the symbols:

Addition +	Subtraction −
Multiplication *	Division /
Exponentiation **	

** is considered to be a single symbol representing the operation 'to raise to a power'. It will be seen later that it can never be confused with or interpreted as the multiplication symbol *. We shall consider at this stage the operands already introduced – that is real and integer variables and constants. Expressions of the following form can now be written:

(*i*) X+Y	(*ii*) VOLTS*AMPS	(*iii*) THETA/3.14159
(*iv*) CHI**2	(*v*) 4.0*A*C	(*vi*) NOFUN/2

One important difference between these FORTRAN expressions and mathematical expressions is that all operators must be explicitly stated. Whereas in mathematics AC (meaning A times C) could be written, in FORTRAN this must always be written as A*C. If the multiplication operation were not stated explicitly AC would be indistinguishable from the FORTRAN variable named AC.

Another important rule which applies to arithmetic expressions in FORTRAN is that no two operators can be juxtaposed. This means that an expression of the form NOFUN/−2 is not allowed, since the operators / and − follow each other immediately. In a case such as this the operators must be separated by the use of parentheses and the expression written in the form NOFUN/(−2). It follows from this that ** can never be used to denote two multiplication operators in succession, and therefore there will never be any ambiguity in the use of ** as an exponentiation symbol.

In the early FORTRAN compilers a further restriction was applied which has, in general, been removed in more recent compilers. All

operands in any arithmetic expression had to be of the same type – that is the expression had to consist of either all integer constants and variables (yielding an integer result) or all real constants and variables (yielding a real result). However, in some FORTRAN IV compilers the operands in an expression can be of any type, and *mixed mode arithmetic* may be performed. The nature of the result which such a mixed expression yields will be discussed later in this section. Note that mixed arithmetic is not allowed in ASA FORTRAN.

First, however, the way in which an arithmetic expression is evaluated must be considered. The examples of arithmetic expressions given so far in this chapter have been of a simple kind. When longer and more complicated expressions are used care must be taken to ensure that no ambiguity is introduced into the meaning of the expression. For example, does the expression

X−Y+Z

mean 'subtract Y from X and add Z to the result' or does it mean 'add Y to Z and subtract this result from X' (these would produce the entirely different results +17 and −23 for X=2, Y=5 and Z=20)? Similarly, does

VOLTS*AMPS**2

mean 'multiply VOLTS by AMPS and square the result' or does it mean 'square AMPS and multiply the result by VOLTS'? This ambiguity is resolved both in mathematical and in FORTRAN expressions by the use of brackets. Thus X−Y+Z is written as (X−Y)+Z or X−(Y+Z) according to which is intended. Similarly, VOLTS*AMPS**2 is written as either (VOLTS*AMPS)**2 or VOLTS*(AMPS**2).

However, if algebraic brackets are not used in an arithmetic expression the FORTRAN operators are given an implicit priority specifying the order in which the operations will be performed. This hierarchy of operators is as follows:

All exponentiations are performed first.
All multiplications and divisions are performed second.
All additions and subtractions are performed last.

Thus VOLTS*AMPS**2 is equivalent to VOLTS*(AMPS**2) – that is the exponentiation, which is of greater priority than the multiplication, is performed first. Similarly,

P/Q**2+X/Y+Z**2

is identical in meaning to

(P/(Q**2))+(X/Y)+(Z**2)

It can be seen, however, that multiply and divide have the same priority and add and subtract have the same priority. This raises the question

of how an expression which consists entirely of operators of the same priority is evaluated. The answer is that the expression is evaluated in order from left to right. Under this scheme X—Y+Z is equivalent to (X—Y)+Z. Similarly

X/Y*Z+P*Q/R*S/T

is equivalent to

((X/Y)*Z)+((((P*Q)/R)*S)/T)

or in mathematical notation

$$\frac{xz}{y}+\frac{pqs}{rt}.$$

An important exception to this left-to-right rule occurs when an expression consists entirely of exponentiations. In this case the operations are performed from right to left. That is I**J**K means 'raise J to the power K and raise I to this resulting power', i.e. it is equivalent to I**(J**K).

It is important to remember that the presence of parentheses overrides this natural order of priority. The use of parentheses in complicated arithmetic expressions, even when they are not absolutely necessary, is often advisable not only for the sake of clarity but also as a precaution to ensure that the way in which the expression is actually evaluated is the same as that intended.

Parentheses used in this way in arithmetic expressions denote only grouping. They do not imply multiplication. The presence of an operator is essential always. Therefore (X+Y) (P+Q) *must* be written as (X+Y)*(P+Q).

Let us now return to consider the question of mixed expressions. In the case of operand – operator – operand triads the following combinations are possible:

(*i*) A/B both operands real
(*ii*) I/J both operands integer
(*iii*) A/J first operand real, second integer
(*iv*) I/B first operand integer, second real.

For A = 5.5, B = 2.0, I = 3 and J = 2 these expressions have the following values

(*i*) 2.75
(*ii*) 1
(*iii*) 2.75
(*iv*) 1.5

In case (*i*) no problems arise. The operands are both real, and therefore the resulting value of the expression is a real number. In case (*ii*)

both the operands are integer, and therefore the result of the operation is an integer. So although the actual result of dividing 3 by 2 is 1.5, when this division is performed in integer mode the value returned is the integral part only, i.e. 1 (note that it is *not* rounded up to 2). In cases (*iii*) and (*iv*) the expressions are mixed. Since real and integer numbers are held in different forms in the computer, arithmetic operations cannot be performed directly between them. Thus if an expression is mixed the integer number is first converted to a real number and then real arithmetic is performed. Hence I and J will be converted to the real numbers 3.0 and 2.0 respectively and expressions (*iii*) and (*iv*) will be carried out as if both operands were real and will yield real results.

This principle holds in more complicated expressions. An expression of the form.

R+S*I/J+N/3

will be computed as follows. S and I will be multiplied together, giving a real result (I having first been converted to a real number). J will then be converted to a real number and divided into the result of S*I to give a real result. This will be added to the real number R, giving a real result. Finally, N/3 will be evaluated as an integer expression giving an integer result. This will then be converted to a real number and added to the previous result. The whole expression will produce a real answer. For R=10.6, S=3.5, I=5, J=2 and N=7 the result will be

$$\begin{aligned} \text{R+S*I/J+N/3} &= 10.6+3.5*5.0/2.0+7/3 \\ &= 10.6+8.75+2.0 \\ &= 21.35. \end{aligned}$$

Note the effect of the truncation in the integer division N/3.

2.2 System functions

There are certain mathematical functions which are in such frequent demand by programmers that to save each person from having to write a program to calculate them they are incorporated in the FORTRAN compiler. These functions are generally referred to as system functions. If a programmer wishes to calculate the square root, sine or cosine of a variable X he need only write down the expressions SQRT(X), SIN(X) or COS(X) at the appropriate point in the program and the system function will return the required value. SQRT, SIN and COS are the function names and X is the *argument*, which must be enclosed in parentheses. The argument can be any expression and not just a single variable. So the square root of b^2-4ac can be found by simply writing

SQRT(B**2—4.0*A*C). The system functions provided depend upon the particular installation, but the following should always be available:

SQRT	finds the square root of a real argument.
SIN	finds the sine of an angle (real argument in radians).
COS	finds the cosine of an angle (real argument in radians).
ATAN	finds the arctangent (real argument).
EXP	finds the exponential of a real argument.
ALOG	finds the natural logarithm of a real argument.
ABS	finds the absolute value (or modulus) of a real argument.

Many more than these are usually provided. The programmer must take care that none of his invented variable names is the same as the name of one of the standard system functions.

2.3 Arithmetic assignment statements

By far the most common use of arithmetic expressions in FORTRAN is in the arithmetic assignment statement. This statement changes the value of a single variable on the left-hand side to the value of the expression on the right-hand side, i.e. it is of the general form

VARIABLE = EXPRESSION

There is an important fundamental difference between the arithmetic assignment statement and the very similar algebraic equation. In FORTRAN the statement X=Y+Z is an instruction to the computer to 'take to an accumulator the contents of the storage location named Y, add to this the contents of the storage location named Z and put the result in the storage location named X'. By this definition the following statement can be written quite legitimately

X=X+1.0

This would be nonsense as an algebraic equation, but as an assignment statement in FORTRAN it means 'take the present contents of the storage location X, add 1 to this number and put the answer back in location X, overwriting anything which is already present in this location'. Thus if X is 5.0 when this statement is encountered its value will be changed to 6.0, and whenever X is used again at a later stage in the program it will have the value 6.0 unless it is changed by another assignment statement. It is always important to have in mind this basic difference between FORTRAN and mathematics. The equals sign in an assignment statement has the meaning 'is set equal to', whereas in mathematics it has the meaning 'is the same magnitude as' (this latter meaning of equals can also be obtained in FORTRAN by using the

logical operator .EQ. which will be described in the next chapter). It is now immediately apparent why the left-hand side of an arithmetic assignment statement can only be a single variable and not an expression. It is obviously meaningless to write something of the form

```
A-B=P**2+Q**2
```

As already discussed, the expression on the right-hand side of an arithmetic assignment statement can have a real or integer value, but what of the variable on the left-hand side? In general, if the expression on the right-hand side is real or integer, then the left-hand side will be a real or integer variable correspondingly. However, there is nothing illegal about writing a statement of the form:

```
REAL VARIABLE = INTEGER EXPRESSION
```

or

```
INTEGER VARIABLE = REAL EXPRESSION
```

If the result of a real expression is assigned to an integer variable, then truncation will occur before assignment takes place and any fractional part will be lost. On the other hand, if an integer result is assigned to a real variable it is first converted to a real result (with the fractional part equal to zero) before the assignment. Consider a statement of the form

```
X=A/B
```

If A is 9.0 and B is 5.0, then X will be given the value 1.8. However, if the statement had been of the form

```
M=A/B
```

M would have been assigned the value 1 (i.e. any fractional part is ignored). So it can be seen that statements of these forms could introduce considerable error into a calculation if they were written without due care and, moreover, since they are grammatically quite correct, they would not be detected by the FORTRAN compiler.

The following examples are all correct FORTRAN assignment statements:

(*i*) `ZETA=0.69*BG`

(*ii*) `CHI2T=ATERM-2.0*BTERM+CHI2PT+CHISR`

(*iii*) `CO7=1.0/(CO3**2+CO4**2)`

(*iv*)
```
RESULT=EXP(-3.14159*ETA)*SQRT((6.28*ETA)
                                /(1-EXP(-6.28*ETA)))
```

(*v*) `POWER=EXP(X*LOG(E))`

Example (*iv*) shows the use of parentheses (*a*) as brackets enclosing the argument of a function and (*b*) as algebraic brackets. Example (*v*)

illustrates that the argument of a function can be an expression which itself contains another function.

Further examples of arithmetic assignment statements can be seen in lines 7, 10, 11, 12 and 13 of Figure 1.3.

The following sequence of assignment statements finds the sum, the arithmetic mean and the product divided by the sum of 5 integers:

```
NUM1 = 5
NUM2 = 62
NUM3 = -11
NUM4 = 22
NUM5 = -6
SUM = NUM1+NUM2+NUM3+NUM4+NUM5
AMEAN = SUM/5.0
PROVSM = NUM1*NUM2*NUM3*NUM4*NUM5/SUM
```

Note that this FORTRAN program would be valid only if the compiler allowed mixed mode arithmetic. In ASA FORTRAN the last line would have to be replaced by

```
PRODCT = NUM1*NUM2*NUM3*NUM4*NUM5
PROVSM = PRODCT/SUM
```

Throughout the rest of the book it will be assumed that mixed-mode arithmetic is available.

2.4 Simple input and output statements

Although the example given at the end of the last section is perfectly correct FORTRAN, it would never be written that way in practice, since it is limited to calculating the sum, the product divided by the sum and the arithmetic mean of the numbers 5, 62, −11, 22 and −6 only. If it were required to calculate these quantities for an entirely different set of numbers a new program would have to be written. This would obviously be time-consuming and wasteful, particularly for large programs. The programmer therefore writes his program in general terms and provides a separate set of data on which it can operate. Thus when he wishes to perform the same sequence of operations on an entirely different set of numbers his program can remain unchanged and he need only provide new data. But the program must now be equipped with some mechanism by which the data can be read into the computer and the required results subsequently printed out. This is done by means of an INPUT/OUTPUT (or I/O) statement. The two most important I/O statements are the READ and WRITE statements,

and since these two are sufficient for our present purposes, we shall here discuss only these. A full description of the FORTRAN I/O system is given in Chapter 5.

FORTRAN is a card-oriented language – that is the program and data are fed into the computer in the form of punched holes on cards rather than on punched tape. Before the READ and WRITE statements can be discussed further the layout of data on a card must be considered. Figure 2.1 (*a*) shows a typical data card for a FORTRAN program. The card consists of 80 columns and 12 rows and every character is represented by a unique pattern of holes in each column. So 80 characters can be punched on any one card. Let us suppose that we have a program which requires numerical values to be assigned to an integer variable I, a real variable P, a real variable Q and an integer L. We must therefore punch on a card four numbers which match up in type with the names to which they are assigned. Thus any legitimate form of integer constant can be punched to correspond to an integer variable and any real constant to correspond to a real variable (see Section 1.3 for a discussion of valid numerical constants). In addition, we must specify in the program the particular form the number takes, i.e. how many columns it occupies, whether it is an integer or real constant and what sort of real constant it is. In order to do this every READ and WRITE statement has associated with it a FORMAT statement which is given a number by which it can be identified. The READ and WRITE statements are of the general form:

READ(m_1,n_1)*list* WRITE(m_2,n_2)*list*

where m_1 and m_2 are positive integer constants which represent the logical numbers associated with input and output devices by means of which data are to be transmitted to and from the computer. The values of these two constants depend very much on the computer used, but throughout this book m_1 will be taken as 5 and m_2 as 6; n_1 and n_2 are the numbers of the FORMAT statements associated with the READ or WRITE statement, and *list* represents a list of FORTRAN variable names (separated by commas) which have to be given the numerical values punched on the card.

The FORMAT statement consists of the word FORMAT followed by a list of *format specifications* enclosed in parentheses. At the moment we shall consider the input of numbers only, and for this only two format specifications (I and F) are needed. The I format specification has the form I*n*. This specifies that the number to be read from the data card is an integer constant covering *n* columns of the card (including the + or − sign). Similarly, a real number (in fixed-point and not floating-point form) is specified by the format F*n.m*, where *n* is the total width of the field, i.e. the total number of column positions the

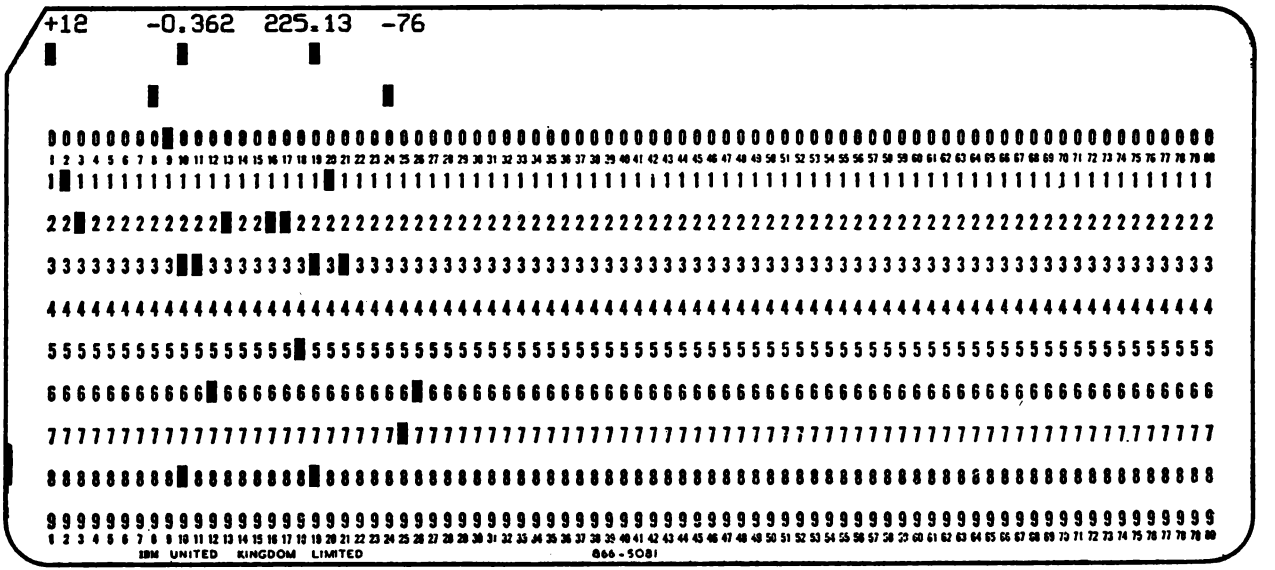


Figure 2.1 (*a*) A typical FORTRAN data card

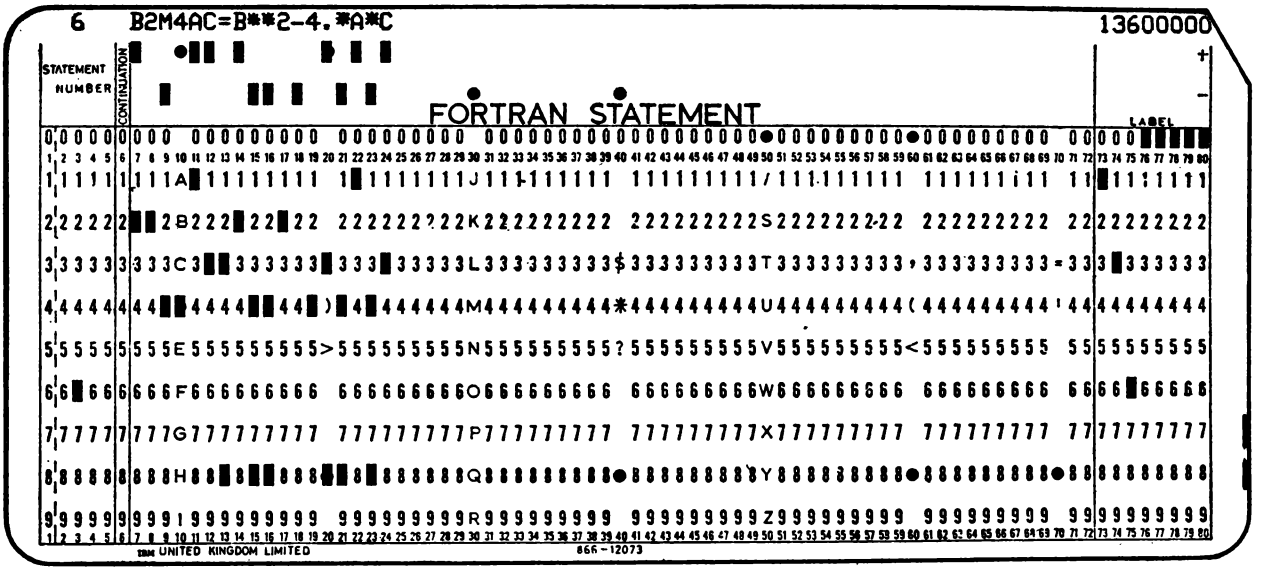


Figure 2.1. (*b*) A typical FORTRAN statement card

Reproduced by Courtesy of International Business Machines Corporation

number takes up on the card, including the decimal point and the + or − sign, and *m* represents the number of digits after the decimal point.

To read the numbers punched on the card in Figure 2.1 (*a*) and to assign them to I, P, Q and L, a statement of the form

```
      READ(5,10) I,P,Q,L
   10 FORMAT(I3,F10.3,F8.2,I5)
```

must be written.

The example given at the end of the previous section can now be written in more general terms as follows:

```
      READ(5,20)NUM1,NUM2,NUM3,NUM4,NUM5
   20 FORMAT(I5,I3,I2,I5,I2)
      SUM=NUM1+NUM2+NUM3+NUM4+NUM5
      AMEAN=SUM/5.0
      PROVSM=NUM1*NUM2*NUM3*NUM4*NUM5/SUM
      WRITE(6,21)SUM,AMEAN,PROVSM
   21 FORMAT(F9.1,F8.4,F10.4)
```

Any set of numbers can then be read into this program provided the layout on the card corresponds to the FORMAT statement 20, i.e. provided there are 5 integer numbers punched on the card occupying 5 columns, 3 columns, 2 columns, 5 columns and 2 columns respectively (including any sign).

Note that the FORMAT statement is a *non-executable* statement. Its function is to provide information to the compiler at compile time. It is *not* translated into machine language instructions to be obeyed at execution time. For this reason a FORMAT statement can appear at any point in a program without it in any way affecting the flow of the logic of that program.

2.5 Program layout

In the previous section the layout of data on cards has been discussed, and it is worth while considering at this stage the way in which the FORTRAN program itself is punched on cards. From the FORTRAN compiler's point of view the card containing the source statement is divided into four distinct regions. These are columns 1 to 5, column 6, columns 7 to 72, columns 73 to 80.

If a letter C is punched in column 1, then this tells the compiler that what follows on this card is purely comment and is not part of the program to be translated. It is thus completely ignored by the compiler.

This facility is provided for the convenience of the programmer so that he can document his program in such a way that other programmers can understand it. Otherwise columns 1 to 5 are reserved for statement numbers, i.e. FORMAT statement numbers and labels (which are to be discussed in Chapter 3) are placed anywhere in these columns. The FORTRAN statement itself is placed between columns 7 and 72. An example of a FORTRAN statement card is shown in Figure 2.1 (*b*). Statements are punched one per card, and two statements on a card are not allowed. It is probably appreciated that a very long arithmetic assignment statement may require more space than is available on one card. To solve this problem *continuation cards* are allowed and the statement can be continued from one card to the next. The fact that a statement punched on a card is simply a continuation of a previous statement is indicated by punching *any* character in column 6 (except ϕ or blank). Apart from this, column 6 is not used. A total of 9 continuation cards (commonly numbered 1 to 9 in column 6) is usually allowed, but this varies from installation to installation.

Anything punched from column 73 to the end of the card is ignored by the compiler. These columns are generally used for numbering the cards. Many programmers have experienced the tedium of sorting into correct order a deck of unnumbered cards which have been accidentally dropped!

It must be stressed that the significance of the different regions on a card is only meaningful to the FORTRAN compiler and bears no relation to the layout of numbers on a data card. This is entirely under the control of the programmer, who has the whole card field at his disposal. Note also that although spaces are significant in data, spaces within a FORTRAN statement are ignored by the compiler.

2.6 The STOP and END statements

Sufficient information has been given in this book so far for a complete FORTRAN program to be written. However, before such a program could actually be run on a computer two further pieces of information have to be supplied. The last statement of any FORTRAN source program must be a message telling the compiler that this is the end of the text to be compiled. This is the function of the END statement. The END statement is a non-executable statement – that is it merely gives information to the compiler and is not placed in the object program as a statement to be obeyed at execution time.

A FORTRAN program must also contain a statement which *is* executable and which when obeyed will cause execution of the object program to be terminated. This is the STOP statement.

The use of these two statements can be seen in lines 16, 20 and 21 of the program shown in Figure 1.3.

EXERCISES 2

(1) Express the following algebraic equations as FORTRAN arithmetic assignment statements:

(*i*) $$a = \frac{3xy^2(z+1) + \frac{1}{2}yz}{1 + x + x^3}$$

(*ii*) $$b = \frac{3}{2}x(x-1)\,[7y - \log_e(\cos x)]^{\frac{1}{2}}$$

(*iii*) $$i = \frac{k}{j}\left[10^2 x - \frac{1}{3.0}\frac{(k^2)}{j}\right]$$

(*iv*) $$z = \frac{5x^2[\cos^3(x^2 - y^2) + \tan^{-1}(x\cos x)]^{\frac{1}{2}}}{(e^{x+1}e^{y+1} + 1)}$$

Assume that all variables beginning with the letters *i* to *n* are to be represented by FORTRAN integer variables and that all others are real variables.

Calculate the value assigned to the left-hand side of the FORTRAN statement equivalent to (*iii*) above for $j=3$, $k=7$ and $x=0.5$.

(2) Write a program to calculate the total interest earned by investing a sum of money at a fixed interest rate for a given number of years using the expression

$$A = P\left(1 + \frac{r}{100}\right)^n$$

where *P* is the initial capital, *r* is the percentage rate of interest, *n* is the number of years for which the sum is invested, and A is the total capital after *n* years. The program should have as its data:

(*i*) the sum invested (in pounds, shillings and pence),
(*ii*) the rate of interest, *r*,
(*iii*) the total number of years, *n*,

and should output the yield both in pounds, shillings, and pence and in dollars and cents. Assume that the exchange rate is 2.4 dollars to the pound.

CHAPTER THREE

Transfer of control

3.1 Introduction

It has been apparent in what has been said so far that the computer will, unless otherwise instructed, obey a sequence of instructions in the order in which they appear in the program. However, FORTRAN provides a set of instructions which when obeyed cause a jump to some other part of the program. These are known as *transfer of control* instructions, since they cause computer control to be passed to some statement other than the one immediately following. Transfers of control may be conditional or unconditional – that is control may be transferred either if some condition is true or regardless of any condition.

3.2 The GO TO statement

This is an unconditional transfer of control statement. It is of the general form

GO TO *n*

where *n* is the statement number to which control is to be transferred. Any executable statement may be preceded by a statement number which must be an unsigned integer of up to 5 decimal digits (punched in columns 1 to 5 of the card), but no two statements must be given the same number. An executable statement is any statement which is translated into an instruction to be obeyed at execution time as opposed to one which simply provides necessary information to the compiler. The executable statement to which control is transferred may occur at an earlier or later stage in the program and unless otherwise directed, the computer will obey this statement and the ones immediately following it in sequence. Statements can be labelled in any order.

3.3 The arithmetic IF statement

The GO TO statement is very important and frequently used in FORTRAN programs, but by itself it allows little to be done which could not

have been done some other way. For example, a section of program of the form

```
      P=X**2+1.0
      GO TO 10
30    R=P/(Q*S*T)
      GO TO 20
10    Q=X+2.0
      S=X+3.0
      T=X+5.0
      GO TO 30
20    ..... (continue)
```

could be written, but this is obviously not necessary, since the same computation could be more efficiently written

```
      P=X**2+1.0
      Q=X+2.0
      S=X+3.0
      T=X+5.0
      R=P/(Q*S*T)
```

Also, unless the programmer is provided with some transfer of control instruction other than the GO TO statement, he may find that the computer will loop indefinitely round the same section of program. For example, a typical program to sum the series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

might be written:

```
      READ(5,10)X
10    FORMAT(F8.3)
      IFACT=1
      X1=X
      E=1.0
      N=1
20    IFACT=N*IFACT
      E=E+X/IFACT
      X=X*X1
      N=N+1
      GO TO 20
      STOP
      END
```

The reader should satisfy himself that this program will sum the correct series but will never stop looping round the section of the program between label 20 and the instruction GO TO 20.

We obviously require, therefore, some statement which will cause transfer of control if some condition is reached. One such statement is the *Arithmetic* IF statement which is of the general form

$$\text{IF } (e)\ n_1, n_2, n_3$$

where e is any arithmetic expression (except complex expressions to be discussed later) and n_1, n_2 and n_3 are executable statement numbers. The action of the statement is as follows: if the value of the expression in parentheses is negative, then control is transferred to statement number n_1, if it is zero, then control is transferred to n_2 and if it is positive to n_3. The example above can now be written as a finite sequence of instructions to sum, say, the first 11 terms of the exponential series (i.e. for values of N from 1 to 10 but not for N=11).

```
      READ(5,10) X
10    FORMAT(F8.3)
      IFACT=1
      X1=X
      E=1.0
      N=1
20    IF(N=11) 21,30,30
21    IFACT=N*IFACT
      E=E+X/IFACT
      X=X*X1
      N=N+1
      GO TO 20
30    WRITE(6,11)E
11    FORMAT(F12.6)
      STOP
      END
```

The statement labelled 20 will now cause control to be transferred out of the loop when the condition n = 11 is met.

It will be seen that the Arithmetic IF statement can be considered as a three-way switch directing the flow of the program into one of three possible paths (which are not necessarily all the same).

3.4 The computed GO TO statement

Another form of unconditional transfer in FORTRAN is the *Computed* GO TO statement. This is of the general form

$$\text{GO TO } (n_1, n_2, \ldots n_m), i$$

where $n_1, n_2 \ldots n_m$ are executable statement numbers in the program and i is an integer variable. (For completeness it must be pointed out that i cannot be a subscripted integer variable. Subscripts are to be discussed in Chapter 4.) Control is transferred to statement numbers $n_1, n_2, \ldots n_m$ according as the value of i is $1, 2, \ldots m$ respectively. Thus GO TO (50,22,8,65,), IQ will cause control to be transferred to label 50 if IQ=1, label 22 if IQ=2, label 8 if IQ=3, and label 65 if IQ=4.

Note that the value of i must lie somewhere in the range 1 to m. If it is outside this range, then (in the IBM 360 FORTRAN IV being discussed here) the next statement will be executed. In general, however, it cannot be predicted what the program will do.

The following example illustrates the use of the computed GO TO statement. Suppose at some part of a program we wish to obtain the value FX of one of several alternative functions, the required function being selected according to the value of the integer variable NOFUN. The relevant section of such a program for four different functions is given below:

```
      GO TO (10,20,30,40), NOFUN

C CALCULATION OF FUNCTION 1
                                        (x1 - x2**2)**2 + (1 - x2)**2
   10 F1=X1-X2*X2
      F2=1.0-X2
      FX=F1*F1+F2*F2

C NOTE THAT IT IS MORE EFFICIENT TO COMPUTE
C F1*F1 THAN TO COMPUTE F1**2
      GO TO 100

C CALCULATION OF FUNCTION 2
                                        100(x2 - x1**2)**2 + (1 - x1)**2
   20 F2=X2-X1*X1
      F1=1.0-X1
      FX=100.0*F2*F2+F1*F1
      GO TO 100
```

```
C CALCULATION OF FUNCTION 3
                                        100(x1 - x2**3)**2 + (1 - x2)**2
   30 F1=X1-X2**3
      F2=1.0-X2
      FX = 100.0*F1*F1+F2*F2
      GO TO 100
C CALCULATION OF FUNCTION 4
                                        (x1 + 2x2 - 7)**2 + (2x1 + x2 - 5)**2
   40 F1=X1+2.0*X2-7.0
      F2=2.0*X1+X2-5.0
      FX=F1*F1+F2*F2
  100 . . . (continue)
```

It can be seen from this that, just as the arithmetic IF statement can act as a three-way switch in a program, so the computed GO TO statement can act as an m-way switch.

3.5 The assigned GO TO statement

For completeness, a further form of the GO TO statement (which is not frequently used) is mentioned here. It is known as the *assigned* GO TO statement, since it consists of a GO TO statement used in conjunction with an ASSIGN statement. These two statements are of the general form

```
ASSIGN i TO m
. . . . . .
. . . . . .
GO TO m, (n1, n2, n3, . . . nk)
```

where i, n_1, n_2, . . . n_k are executable statement numbers and m is an integer variable (which must not be subscripted. See Chapter 4 for a discussion of subscripted variables.)

The statement causes control to be transferred to the statement labelled n_1 or n_2 . . . or n_k according as m has been currently assigned the label n_1 or n_2 . . . or n_k. Thus

```
ASSIGN 15 TO LABEL
. . . . . .
. . . . . .
GO TO LABEL, (22,6,15,30,1)
```

will cause control to be transferred to the statement labelled 15. Note that the actual value of LABEL is not 15, i.e. the statement ASSIGN 15 TO LABEL is *not* the same as the assignment statement LABEL=15.

3.6 The logical IF statement and logical expressions

There is a need in many programs for a further kind of conditional transfer of control statement. For example, it may be required to take some action if one expression is less than, greater than or equal to some other expression – that is the logical relationship between two expressions must be tested and the required action taken if the test is positive. To do this, FORTRAN provides the *logical* IF statement, which is of the general form

IF (*lexpr*) *s*

where *s* is any executable statement (other than a DO statement (see later) or another IF statement) which is to be executed if the logical expression *lexpr* is true and ignored if it is false.

The simplest form of logical expression is a single logical variable which can be assigned either a true or a false value. For a discussion of logical constants and variables see Section 7.4.

Another very common form of logical expression (and one frequently used with the logical IF statement) is the *relational expression*. This has the general form

$e_1\ r\ e_2$

where e_1 and e_2 are arithmetic expressions (of type real or integer) being compared by one of the following *relational operators*, *r*

.EQ.	Equal to (=)
.LT.	Less than (<)
.GT.	Greater than (>)
.LE.	Less than or equal to ($\leqslant$)
.GE.	Greater than or equal to ($\geqslant$)
.NE.	Not equal to ($\neq$)

The mathematical equivalents of these operators are given in brackets. Note that the FORTRAN operator .EQ. has the same meaning as '=' in an algebraic equation. The symbols for the relational operators are placed between full stops to distinguish them from the ordinary FORTRAN variable names EQ, LT, etc.

It must be stressed that only expressions of type real or integer can be combined with these relational operators.

Examples of valid logical IF statements using relational expressions are:

```
IF(X1.GT.X2) GO TO 55
IF(X1.LT.0.01) F=0.6*X1+0.8
```

The second example illustrates that the logical IF statement need not necessarily be a transfer of control statement, although it is frequently used as this (see lines 6 and 9 of Figure 1.3 as further examples of two different ways of using the logical IF statement).

More complicated logical expressions can be built up using the *logical operators*

.AND. .OR. .NOT.

to combine logical constants or variables, or relational expressions.

The meanings of these logical operators are best illustrated by the following examples:

(i) IF(Y1.GT.Y2.AND.ICOUNT.LT.20) GO TO 101

This causes a transfer of control to statement 101 if *both* the conditions 'Y1 greater than Y2' *and* 'ICOUNT less than 20' are true. If one (or both) is false, then the GO TO statement will not be executed and control will pass to the next statement in the program.

(ii) IF(P.EQ.Q.OR.X.LT.1.0) GO TO 27

In this case the GO TO statement will be executed if *one or both* of the conditions 'P equals Q' and 'X less than 1.0' is true. Only when they are both false will the GO TO statement be ignored.

(iii) IF(.NOT.X1.GT.X2) GO TO 60

has the same effect as the statement

IF(X1.LE.X2) GO TO 60

i.e. the .NOT. operator reverses the truth value of whatever follows. Therefore the transfer of control statement will be executed if 'X1 greater than X2' is *not* true.

By using combinations of logical and relational operators, complicated logical expressions can be written such as

(iv) IF(A.GT.3.0.AND..NOT.C.EQ.D.OR.B.LT.C) GO TO 10

This illustrates two points. Firstly, two logical operators may not appear in sequence unless the second one is the .NOT. operator. Secondly, ambiguities can be introduced into logical expressions. The logical expression given in example (iv) could mean either

IF(A.GT.3.0.AND..NOT.(C.EQ.D.OR.B.LT.C)) GO TO 10

or

IF((A.GT.3.0.AND..NOT.C.EQ.D).OR.B.LT.C) GO TO 10

This shows that, just as in the case of arithmetic expressions, ambiguities can be removed by the use of parentheses. Similarly, in case the ambiguity is not resolved by parentheses, the relational and logical

operators are given an implicit order of priority. This hierarchy is shown (in relation to the arithmetic operators) below:

Exponentiation	1st priority
Multiplication and division	2nd priority
Addition and subtraction	3rd priority
The relational operators	4th priority
.NOT.	5th priority
.AND.	6th priority
.OR.	7th priority

On this scheme example (iv) has the meaning

```
IF((A.GT.3.0.AND..NOT.C.EQ.D).OR.B.LT.C) GO TO 10
```

since .AND. is more binding than .OR. .

EXERCISES 3

(1) If I=1, J=2, K=3, L=4, and M=5 evaluate the truth value of the following logical expressions:

(*a*) J.GT.K.OR.L.EQ.M−I
(*b*) I+J.NE.K*L.AND..NOT.M.GT.3.OR.K.LT.2
(*c*) .NOT.J−3.EQ.−1.OR..NOT.(M.LT.50.AND.K.EQ.I+1)

(2) What is the final value of J in the following short section of program?

```
      ASSIGN 5 TO L
      J=1
      I=2
    1 K=J+I
    5 ASSIGN 1 TO L
      J=J+1
      IF(J.GT.I.OR.K.EQ.10)GOTO 13
      GOTO (1,5,3,13),J
   12 I=I+1
    3 IF(K−3*I)5,13,7
   13 ASSIGN 12 TO L
      IF(.NOT.I.LT.4) I=0
      IF(K.EQ.6)GOTO L, (1,5,12)
      GOTO 1
    7 STOP
      END
```

(3) Using the flow chart shown in Figure 1.2 as a guide, write a FORTRAN program to sum the squares of the first N integers.

(4) Write a program to evaluate the square root of a number which is to be read in as data. The program should use the formula

$$b = \tfrac{1}{2}\left(\frac{x}{a} + a\right)$$

where b is a better approximation to the square root of x than a. The first approximation should be taken as $\frac{1}{2}x$ and the procedure should be repeated until the difference between a and b is less than 10^{-6}. [Note that this is the sort of program which is called into operation when a reference to a standard system function (in this case SQRT) is found in a FORTRAN program.]

CHAPTER FOUR

Subscripted variables and the DO statement

4.1 Introduction

Let us consider again the short FORTRAN program segment (see Section 2.4) to input a set of numbers and calculate the sum, the product divided by the sum and the arithmetic mean. The numbers are held in a series of five locations referred to by the five distinct names NUM1, NUM2, . . . NUM5, which are all of the same type and on which we wish to perform the same operations. It would be more convenient if instead of referring to the cells by five distinct names we could define one name, NUM, to which would be allocated a sequence of five locations in the computer. Any particular one of these locations would then be selected according to the value of a subscript. Thus NUM1, NUM2, NUM3, NUM4 and NUM5 could be referred to as NUM(1), NUM(2), NUM(3), NUM(4) and NUM(5). The difference between these two forms of notation at the moment may seem trivial, but the full power of this facility will become apparent later in this chapter.

4.2 The DIMENSION statement

The FORTRAN source program must provide some means of indicating to the computer how many locations are to be allocated to the subscripted variable named NUM. This information is provided by means of the DIMENSION statement, which is of the general form

DIMENSION $name_1(m_1)$, $name_2(m_2)$. . . . $name_n$ (m_n)

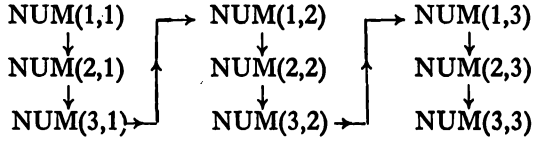
where $name_1$, $name_2$, . . . $name_n$ are allowable FORTRAN variable names and m_1, m_2, . . . m_n each represent a list of unsigned integer constants, separated by commas, representing the maximum value of each subscript. m_1, m_2, ... m_n can also be lists of integer variables in certain circumstances and this will be discussed further in Chapter 6. Therefore in the particular example being considered the required DIMENSION statement is of the form

DIMENSION NUM(5)

This informs the compiler that the following program wishes to refer to five distinct sequential locations all identified by the variable NUM.

Note that the DIMENSION statement (like the FORMAT statement discussed in Section 2.4) is a non-executable statement.

The complete set of locations referred to by NUM is known as an *array*, and any particular location (e.g. NUM(3)) is referred to as an *element* of that array. All elements of an array are of the type specified by the name of the array. NUM is a one-dimensional array, since the particular element required is identified by the value of one subscript only. However, there is no reason why arrays having two or more dimensions cannot be used and the required element selected according to the value of two or more subscripts. Thus nine consecutive locations could be referred to by means of a one-dimensional array NUM(1), ... NUM(9) or a two-dimensional array NUM(1,1), NUM(2,1), NUM(3,1), NUM(1,2), NUM(2,2), NUM(3,2), NUM(1,3), NUM(2,3), NUM(3,3). The latter can be thought of as an array composed of horizontal rows and vertical columns. The value of the first subscript refers to a particular row and of the second to a particular column:



So NUM(2,3) refers to that element in the second row and the third column. The arrows indicate the order in which the elements are stored in the computer. They are stored in such a way that the first subscript varies the most rapidly so that the elements NUM(3,1) and NUM(1,2) are in consecutive locations.

The DIMENSION statement defining this array would be

```
DIMENSION NUM(3,3)
```

A maximum of seven dimensions is allowed in the IBM 360 version of FORTRAN IV, although many compilers have a much lower maximum than this (including ASA FORTRAN IV, which allows a maximum of three dimensions). For two or more dimensions the subscripts must be separated by commas.

It is important that the information concerning the number of locations to be assigned to a particular variable should be given to the compiler before any actual reference is made to these locations in the active part of the program. Therefore the DIMENSION statement must be placed before any statement operating on a subscripted variable. It is common practice to place the DIMENSION statement at the head of the program.

4.3 Types of subscript

It has been shown so far that subscript notation can be used in FORTRAN, but the real advantage of using the array NUM(1), NUM(2), ... NUM(5) instead of the FORTRAN variables NUM1, NUM2, ... NUM5 is not immediately obvious. However, if the subscripts themselves are allowed to be variables or expressions, then the power of this facility is extended considerably. If it is required to perform the same sequence of operations on the elements of an array it is only necessary to specify the computation once and arrange for computer control to pass through the sequence of instructions many times, each time selecting the next element of the array by changing the value of the subscript. This will be clarified by consideration of the FORTRAN program segment to find the sum, the product divided by the sum and the arithmetic mean of a set of numbers. This program can now be written as follows:

```
      DIMENSION NUM(5)
      PRODCT=1.0
      SUM=0.0
      I=1
10    READ(5,20) NUM(I)
20    FORMAT(I3)
      SUM=SUM+NUM(I)
      PRODCT=PRODCT*NUM(I)
      I=I+1
      IF(I.LT.6) GO TO 10
      AMEAN=SUM/5.0
      PROVSM=PRODCT/SUM
      WRITE(6,21) SUM, AMEAN, PROVSM
21    FORMAT(F9.1,F8.4,F10.4)
      STOP
      END
```

The reader should work methodically through this example and satisfy himself that the calculation performed here is identical to that given in the program at the end of Section 2.4. It will be seen that computer

control passes five times through the section of the program between statement 10 and the statement IF(I.LT.6) GO TO 10. Each time, the value of I has been increased by 1, so a new element of the array NUM is added into the location SUM.

The power of this facility, when instead of five numbers there are hundreds, should be striking.

It was stated earlier in this section that subscripts can be expressions. The following rules apply to the formation of subscripts:

(*i*) A subscript may be any allowable *arithmetic* expression which has an integer result. If the expression has a real result, then it will be truncated and converted to integer.

(*ii*) Subscripts can contain function references and subscripted names.

(iii) The value of any subscript must never be zero. It must lie between 1 and the maximum specified in the DIMENSION statement.

Note that in this version of FORTRAN the types of subscript allowed are very flexible. Many versions of FORTRAN (in particular ASA FORTRAN) allow only very restricted integer expressions to be used as subscripts, and this should be checked.

4.4 The DO statement

In the example in the previous section it was arranged for computer control to pass through the same section of program many times by: (i) setting the initial value of some counter, i.e. I=1; (ii) incrementing the counter each time the calculation was performed, i.e. I=I+1; and (iii) using a conditional jump statement to return control to the beginning of the section concerned if the calculation had not been performed the required number of times.

In FORTRAN these three operations can be performed by one instruction – the DO statement. This is of the general form:

$$\text{DO } n\ i = m_1,m_2,m_3$$

where n is the number of some executable statement at a later point in the program and i is a non-subscripted integer variable representing the counter to be incremented (e.g. I), m_1, m_2 and m_3 are all unsigned integer constants or non-subscripted integer variables (with values greater than zero) such that m_1 represents the initial value of the counter (e.g. 1), m_2 represents the final value (e.g. 5) and m_3 represents the step by which the counter is to be incremented each time through the loop

(note that a negative step with m_2 less than m_1 is not allowed). If m_3 is not explicitly stated it is taken to be 1. In this case the statement is of the form

DO n i=m_1,m_2

The DO statement automatically causes execution of all the statements following it up to *and including* the statement labelled n for values of i from m_1 in steps of m_3. Thus i is set equal to m_1 and the required computation is performed; i is then increased to m_1+m_3 and the calculation repeated. This is continued until i becomes greater than m_2, when control is transferred out of the loop. The set of statements which is actually executed within the loop is known as the *range* of the DO statement. The counter i is sometimes referred to as the *index* of the loop and sometimes as the DO variable. Its value on exit from the loop is, in general, undefined.

The program segment in the previous section can now be rewritten using the DO statement.

```
      DIMENSION NUM(5)
      PRODCT=1.0
      SUM=0.0
      DO 30 I=1,5
      READ(5,20) NUM(I)
   20 FORMAT(I3)
      SUM=SUM+NUM(I)
   30 PRODCT=PRODCT*NUM(I)
      AMEAN=SUM/5.0
      PROVSM=PRODCT/SUM
      WRITE(6,21) SUM,AMEAN,PROVSM
   21 FORMAT(F9.1,F8.4,F10.4)
      STOP
      END
```

Again the reader should satisfy himself that this produces the same effect as the program segment in the previous section.

Since m_1, m_2 and m_3 can also be non-subscripted integer variables, this piece of program can be written in a more general way to sum N numbers. Assuming that N cannot exceed 100, say, and reserving 100

locations in the DIMENSION statement, we can now rewrite the program in more general terms as follows:

```
      DIMENSION NUM(100)
      READ(5,20) N
   20 FORMAT(I3)
      PRODCT=1.0
      SUM=0.0
      DO 30 I=1,N
      READ(5,20) NUM(I)
      SUM=SUM+NUM(I)
   30 PRODCT=PRODCT*NUM(I)
      AMEAN=SUM/N
      PROVSM=PRODCT/SUM
      WRITE(6,21) SUM,AMEAN,PROVSM
   21 FORMAT(F9.1,F8.4,F10.4)
      STOP
      END
```

Exit from the loop will automatically take place when it has been completed N times. Control will be transferred to the first statement outside the range of the DO statement. This program is now in its final form, and can be run many times, reading in different values of N followed by a set of N integers.

A conditional jump within the range of the DO statement can also cause control to be transferred out before the cycle is completed. It is, however, illegal to arrange for control to be transferred into the DO range from outside. The reason for this is that the DO statement itself would be by-passed and a loss of necessary information (e.g. the current value of I) would result. The rules governing the transfer of control into and out of DO loops will be summarized in the next section.

Note that the nature of the DO statement is such that the statements within its range will always be executed at least once. Even if the statement is of the form

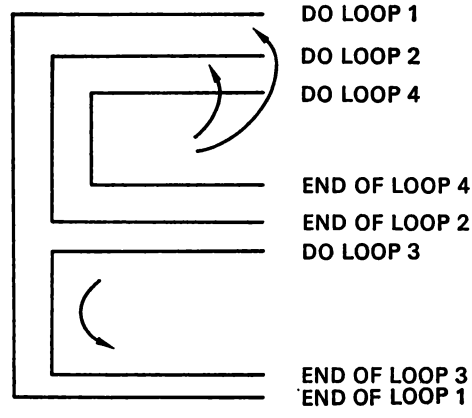
```
DO 10 I=1,1
```

the loop will be performed once for I=1, but when the increment is added it will be found that the current value of I (2) is greater than the final value (1) and no more cycles will be executed.

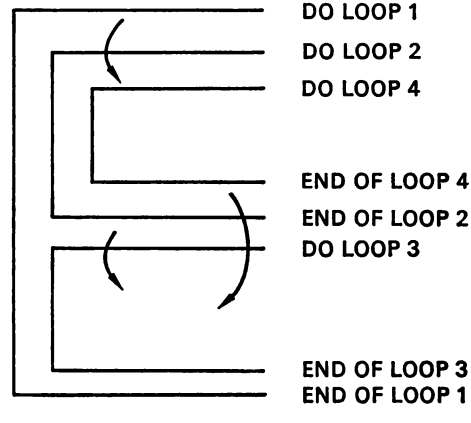
If the DO statement is of the form

```
DO 10 I=1,6,2
```

the loop will be executed for I=1,3,5 but not for I=7, which is greater than the maximum value (6). This illustrates that the index I need never actually be equal to the final value m_2, but execution of the loop will be terminated as soon as it *exceeds* that value.



(a) VALID



(b) INVALID

Figure 4.1. Schematic diagram showing transfers of control within nested DO loops (*a*) valid (*b*) invalid

4.5 A summary of rules applying to the DO statement

(1) The last statement in the range of a DO statement must not be a transfer of control statement of any kind, another DO statement or a logical IF statement. It must be an executable statement.

(2) None of the parameters i, m_1, m_2 or m_3 can be changed by any other statement within the range of a DO statement. They may be changed outside the range provided no transfer is subsequently made back into the range.

(3) An inner DO statement can be used within the range of an outer DO statement. This is generally referred to as a *nest* of DO statements. The DO statements can be nested to any depth. Control can be transferred from the range of an inner DO to the outer range without any restriction, but must never be transferred from an outer range to an inner range. If, in the diagrams of Figure 4.1, [represents the range of a DO statement, then the arrows indicate possible transfers of control. In Figure 4.1 (*a*) the arrows represent valid jumps and in Figure 4.1 (*b*) they represent invalid ones.

The following example illustrates the use of two nested DO statements:

```
      DO 10 I=1,10
      DO 10 J=1,20
10 X(I,J)=0.0
```

4.6 The CONTINUE statement

This is a dummy statement which can be placed anywhere in a program. It is an executable statement, in that it can be labelled and can be used to terminate a DO loop, but logically it has no effect and does not change the sequence of operation of the object program instructions. The statement is simply the word CONTINUE typed in column 7 onwards of a card. It is a particularly useful statement when the range of a DO loop would otherwise terminate on a non-executable statement or on a transfer of control statement, e.g.

```
      DO 20 I=1,10
      X(I)=Y(I)-Z(I+1)
10 IF(X(I).LE.0) GO TO 20
      X(I)=X(I)-Z(I+1)
      GO TO 10
20 CONTINUE
```

EXERCISES 4

(1) What is wrong with the following short program which reads in a list of N integers, computes the sums of alternate numbers (NSMODD, NSMEVN) and forms a new list in which the Ith element, X(I), is the

arithmetic mean of the (I−1)th and (I+1)th elements of the original list (NO)?

```
1     READ(5,1)N
2   1 FORMAT(I4)
3     DO 2I=1,N
4   2 READ(5,3)NO(I)
5   3 FORMAT(I4)
6     DO 4 I=1,N,2
7   4 NSMODD=NSMODD+NO(I)
8     NSMEVN=0
9     DO 5 I=2,N,2
10  5 NSMEVN=NSMEVN+NO(I)
11    WRITE(6,6)NSMODD,NSMEVN
12  6 FORMAT(I6,I6)
13    DO 7 I=1,N−1
14  7 X(I)=(NO(I−1)+NO(I+1))/2
15    X(I)=NO(I−1)
16    DO 8 I=1,N
17  8 WRITE(6,3)X(I)
18    STOP
19    END
```

(2) Write a program to take an array of numbers 7 columns wide and 5 rows deep and to add this array N1 to a similar array N2 to form the sum (array N3). Write this result into an array N4 which is 5 columns wide and 7 rows deep.

(3) Write a program to read a list of N positive real numbers and to calculate and print the largest, the smallest, the mean and the standard deviation using the formula

$$\text{Standard deviation} = \sqrt{\left[\frac{\sum(\text{number-mean})^2}{N}\right]}$$

CHAPTER FIVE

Input and output

5.1 Introduction

An introduction to the FORTRAN INPUT and OUTPUT statements has already been given in Section 2.4, and the reader who is not familiar with the basic principles should read this section before studying the following chapter.

To summarize, a FORTRAN program must provide three necessary pieces of information in order to input or output data. These are: (i) the number of the I/O device used; (ii) the layout or *format* of the data; and (iii) the list of FORTRAN variables to which the data are to be input or from which results are to be output. This information is provided by either a READ or a WRITE statement together with its associated FORMAT statement (some versions of FORTRAN provide I/O statements which do not require FORMAT statements, but these have a specialized use and will not be discussed here).

The READ and WRITE statements are of the general form

$$\text{READ}(m_1, n_1)\ \textit{list} \quad \text{and} \quad \text{WRITE}(m_2, n_2)\ \textit{list}$$

where m_1 and m_2 are unsigned integer constants or non-subscripted integer variables specifying the number of the I/O device (throughout this book m_1 will be taken as 5 and m_2 as 6); *list* is a list of integer or real variable names separated by commas; n_1 and n_2 are unsigned integer constants specifying the FORMAT statement associated with the particular READ or WRITE statement. The FORMAT statement is of the general form

$$n\ \text{FORMAT}(f)$$

where n is an unsigned integer constant representing the FORMAT statement number and f represents a list of *format specifications* separated by commas. It is the main purpose of this chapter to describe in full detail the various forms of the format specifications which can be used in FORTRAN, and it will be seen how the power of the I/O system is increased far beyond that of the simple system already discussed.

Before proceeding with this it is worth pointing out that, in the author's experience, the input of data is the greatest single cause of error in FORTRAN programming. An attempt will be made in the following sections to point out the ways in which errors can arise when using

FORMAT statements, and it is recommended that the reader spends time in thoroughly mastering the principles involved.

5.2 The I/O list

In its simplest form the FORTRAN I/O list consists of a series of integer or real variables separated by commas. On input, this list is scanned and each variable in the list is paired with the next field on the card. The field is specified in the FORMAT statement associated with the READ statement. After one number has been input the next format specification is taken together with the next element of the list. Thus the format specifications and the I/O list keep in step, and numbers are input one after another until the end of the input list is reached. This concept of pairing a particular field with the corresponding element of the I/O list is central to the whole theory of the FORTRAN I/O system.

The examples of I/O statements given in Chapter 2 are very straightforward, and no difficulties should arise. More complicated situations arise when a READ statement and a FORMAT statement do not exactly match. It is often convenient in a FORTRAN program to use one FORMAT statement with several READ or WRITE statements. For example

```
   READ(5,2) I,AREA,J,WIDTH
 2 FORMAT(I5,F8.4,I3,F7.4)
   .....................
   .....................
   READ(5,2)NO,VEL,ICOUNT
   READ(5,2)I,LENGTH,COUNT,WIDTH,VOLUME,RESULT
```

In the first READ statement the I/O list and the format specifications match exactly, and there is no problem. In the second READ statement there are more FORMAT specifications than elements in the list. In this case the first three specifications are matched with the corresponding elements in the I/O list, and the last part of the FORMAT statement is ignored. Therefore a data card punched thus

*bbb*22*b*96.1569*b*58

(where *b* represents a blank or space) will be correctly input with 22 in NO, 96.1569 in VEL and 58 in ICOUNT.

In the third READ statement there are two complications. The first is that there are more elements in the list than there are format specifications in the FORMAT statement. In a situation like this the FORMAT statement is repeatedly scanned from the beginning until the

input list is exhausted, each rescan causing a new data card to be selected. Thus inputting to I, LENGTH, COUNT and WIDTH ends the first scan of the FORMAT statement. A new card will be selected and data will be input to VOLUME and RESULT according to the I5 and F8.4 specifications which appear at the beginning of the second scan of FORMAT statement 2.

The second complication in this case is that the nature of the FORTRAN variables in the list does not always coincide with the corresponding format specification. Thus data are being read into the integer variable LENGTH with a format specification F8.4, into COUNT (real) with an I3 specification (integer) and into VOLUME (real) with an I5(integer) specification. This is rarely faulted on compilation. Inputting an integer to a real variable often results in the integer being changed into floating point form. Inputting a real number to an integer variable will result in the number being truncated. However, this is not always the case, and some versions of FORTRAN will fault this at execution time.

Before more complicated forms of the I/O list are discussed in the next section the concept of a *record* and the way in which it is treated must be considered. On input a record is one data card and on output it is one line of printing. One simple READ statement and associated FORMAT statement refer to one record unless (as occurred in the previous example) the FORMAT statement has to be rescanned. Thus

```
      READ(5,1) I,J,K
1     FORMAT(I5,I8,I7)
```

demands that the numbers to be assigned to locations I, J and K all appear on one card. When the next READ statement is encountered a new card will be read. Suppose it is required to punch the data on three separate cards. Then separate READ statements of the form

```
      READ(5,1)I
      READ(5,2)J
      READ(5,3)K
1     FORMAT(I5)
2     FORMAT(I8)
3     FORMAT(I7)
```

could be written. Every time a READ statement is encountered a new record will be selected. Alternatively, a slash in a FORMAT statement

in place of a comma indicates that a new record is to be taken. The statements

```
READ(5,1)I,J,K
1 FORMAT(I5/I8/I7)
```

would have the same effect as the sequence of statements in the last example.

5.3 The implied DO statement

A subscripted variable can also appear as an element of an I/O list, e.g.

```
READ(5,10) A(I,J), B(I), C(J)
10 FORMAT(F10.5,F8.4,F9.4)
```

Note that this statement is only meaningful if the subscripts I and J have previously been set to the required values so that the correct element of the array can be selected for input. The value of I and J can be read by the same READ statement if desired, but must be placed before any element using them, i.e. to the left in the I/O list. For example,

```
READ(5,11) I,J,A(I,J),B(I),C(J)
11 FORMAT(I5,I5,F10.5,F8.4,F9.4)
```

In this way data can be read into any selected elements of an array. To read data into an entire array it is only necessary to write the name of the array (without subscripts) in the I/O list, and provided the DIMENSION statement for the array has appeared before this READ statement is encountered the entire array will be transmitted. Thus

```
READ(5,12) A
12 FORMAT(F10.5)
```

will read into the entire array data composed of one number per card each with format specification F10.5. Note the importance of dimensioning the array before this statement is encountered. If no DIMENSION statement has previously appeared this statement would just input a single number to the FORTRAN real variable A. Also note that it is illegal to have a subscripted variable (say, A(I)) with the same name as a simple variable (A).

It is sometimes required to read data into a certain section of an array only. To do this FORTRAN provides the implied DO statement. For example, the statements

```
READ(5,13) (A(I),I=1,10)
13 FORMAT(F8.3)
```

will read data (consisting of one number per card) into the first ten elements of the array A. Any section of an array can be transmitted in this way.

```
      READ(5,14) (A(I),I=15,20)
   14 FORMAT(F8.3)
```

will transmit six elements in the body of the array, i.e. elements A(15) to A(20) inclusive.

In a sense, the implied DO statement is analogous to an actual DO statement. The statements

```
      READ(5,13) (A(I),I=1,10)
   13 FORMAT(F8.3)
```

have the same effect as

```
      DO 20 I=1,10
   20 READ(5,13) A(I)
   13 FORMAT(F8.3)
```

In this particular case the effect is identical. Numbers are read, one from each card, into elements A(1) to A(10) of the array A. However, in the second case the READ statement is actually encountered each time round the loop, and hence a new card is automatically selected for each element of the array. So, regardless of the form of the FORMAT statement, this method of reading in an array demands that the numbers are presented one on a card. However, in the first situation this is not the case. The implied DO statement is part of the READ statement. Therefore it is the FORMAT statement alone which determines the form of the input data. The statement FORMAT (F8.3) also demands that the input data be presented one number on a card, but in a later section it will be seen how repeated field specifications can be used to arrange the data as required.

Implied DO loops can be nested. Statements of the form

```
      DO 30 I=1,10
      DO 30 J=1,5
   30 READ(5,14) A(I,J)
   14 FORMAT(F10.4)
```

can also be written

```
      READ(5,14)((A(I,J),J=1,5),I=1,10)
   14 FORMAT(F10.4)
```

This would input data into the two-dimensional array A in the order A(1,1), A(1,2) to A(1,5), A(2,1), A(2,2) to A(2,5) and so on to A(10,1),

A(10,2) to A(10,5). Note the use of commas and brackets in nested implied DO statements.

Ensuring that the elements of the I/O list keep in step with the format specifications in the FORMAT statement is not so straightforward when implied DO statements are used, since the elements are not written out in a string. However, the principle still holds and the FORTRAN programmer should always keep this in mind.

5.4 Format specifications

All format specifications (or codes) consist of two parts: a letter which describes the nature of the data to be input or output and a number which specifies the *field width* of the data (i.e. the number of column positions occupied). For the purposes of this discussion the types of data will be divided into numeric, textual (i.e. the entire character set of letters, digits, punctuation marks, etc.), double precision, and logical fields.

5.4.1 *Numeric fields*

Integers are transmitted by means of the I format code which is of the general form

I*w*

where *w* is the field width; it must include the sign position and any blanks. The + sign may be omitted for positive integers. Decimal points are not allowed at all.

On input, blanks are taken as zero, so care must be taken to position the integer correctly in the field, i.e. an integer must be *right justified* in the field. If the following numbers are punched on a data card (starting at column one)

−1265

+53*bb*

*bb*762

then with a format specification I5 the integers −1265 (correct), +5300 (incorrect), and +762 (correct) would be input. Embedded and leading blanks are also taken as zero.

On output if the number has a field width less than *w*, then the left-most positions will be filled with blanks. If the width is greater than *w*, then either the least significant digits are lost or (on the IBM 360) asterisks are printed out instead of the number. The action depends on the compiler used.

If the number occupies more than *w* positions on input, then only

those digits within the field will be transmitted, and the remainder will be taken as part of the next field.

Real numbers can be transmitted either in fixed point or in floating point form (see section 1.3). The F format specification is used to transmit real numbers without an exponent. It is of the form

F*w.d*

where *w* represents the total field width including the sign, decimal point and spaces; *d* is the number of places after the decimal point. As for the I code, leading, trailing and embedded blanks are taken as zero on input. The decimal point need not be punched on the card; its position will be taken to be that in the format specification. If a decimal point is present, then its position will override that given in the format specification should the two differ.

On output, *d* positions will be printed to the right of the decimal point. If the fractional part has more digits than this, then it will be rounded off. The field width *w* must allow for the decimal point and sign. If the integer part is too large, then asterisks are printed out. If it is too small the number will be preceded by leading spaces.

Suppose the following numbers are punched on a card (starting in column one):

+2539.562
*bbb*63.2*bb*
*bbb*63.200
*b*1.95682573
*b*63.2*bbbb*

then, using a format specification of F9.4, the numbers transmitted will be +2539.562 (correct), +63.200 (correct – note that trailing blanks have no effect in this case), + 63.200 (correct), 1.956825 (incorrect – the extra digits are lost and taken as part of the next field) and +63.2 (correct – note that the actual position of the decimal point overrides that specified in the FORMAT statement).

Real numbers with an exponent are transmitted by means of the E format specification. This is of the form:

E*w.d*

where *w* is the total field width, including the sign, decimal point, exponent and any blanks; *d* specifies the number of digits after the decimal point. On input four positions are reserved for the exponent, so the rest of the number should contain no more than *w*—4 characters. A number with less than four positions in the exponent can be input with this specification. However, since trailing blanks (as well as leading

and embedded blanks) are taken as zero, it is important that the number should be right justified in the field. The decimal point need not be punched, but if it is present its position overrides that given in the format specification. With a format specification of E11.4 the numbers +0.5632E+03, +0.2986E04 and −0.652E−2 will be input as +0.5632E+03 (correct), +0.2986E040 (incorrect – since three characters are read after the E and the number is not right justified, the trailing blank will be read as zero) and −0.652E−200 (incorrect – since the position of the decimal point in the data overrides that specified in the format code, a zero will not be inserted after the fractional part, but two trailing blanks after the exponent will be read as zero; this will be faulted on input, since the exponent is greater than a two-digit number). On output *d* significant digits are printed to the right of the decimal point. If there are more than this in the number, then the fractional part will be rounded. An allowance should be made in *w* for a sign, one digit to the left of the decimal point and an exponent taking four positions.

5.4.2 *Textual fields*

It is often required to transmit textual information as well as numerical data, and the textual field specifications provide the programmer with a means of doing this. The A format specification is of the form:

Aw

and allows any *w* characters of the FORTRAN character set (i.e. letters, digits, punctuation marks, etc.) to be transmitted. The action of the A format specification depends upon the number of characters, *c*, which can be stored in one computer word (see Sections 1.1 and 1.3).

On input, if *w* is less than *c*, then *w* characters will be input and positioned so that they are left justified in the word. The remainder of the word will be filled out with blanks. Suppose a computer word holds 8 characters (represented by 8 boxes in the diagram below), and it is required to input the caption RUN*b*1 with a format specification A5. Then the characters will be held in the word as shown in the following diagram:

R	U	N	*b*	1	*b*	*b*	*b*

If *w* is greater than or equal to *c*, then the *rightmost c* characters will be taken from the card and the word will be filled. Thus if it were required to input the caption SPECTRUM*b*1 with a format specification A10 the 8 characters held in the computer word would be

E	C	T	R	U	M	*b*	1

i.e. the first *w*−*c* characters are ignored.

On output, if *w* is less than or equal to *c*, then the *leftmost c* characters will be printed, i.e. if the contents of the location represented in the first diagram above were output with a format specification of A6, then the characters RUN*b*1*b* would be printed. If *w* is greater than *c*, then *w*—*c* blanks will be printed, followed by *c* characters, i.e. if the contents of the location in the second diagram above were output with a format specification A10, then the characters *bb*ECTRUM*b*1 would be printed.

It can be seen from this that, using the A format specification, textual information can actually be stored in character form in the computer. Frequently it is only required to print out a message to annotate the results, and this can be done without having to hold this message in a FORTRAN variable. Instead a special type of FORMAT specification called the Hollerith specification is used. This is of the form

*w*H

and causes the *w* characters following the H to be transmitted. If the H specification is followed by *w* blanks, then on input *w* characters will be inserted in the FORMAT statement. On output *w* characters are printed. So if it is required to input a caption *b*END*b*OF*b*FIRST*b*DATA*b*SET (22 characters) and then output it the following statements are required:

```
      READ(5,50)
      WRITE(6,50)
50 FORMAT(22Hbbbbbbbbbbbbbbbbbbbbbb)
```

The READ statement causes the 22 characters on the data card to replace the 22 blanks in the FORMAT statement 50. These are then output by the WRITE statement. Note that no I/O list is required in this case, since the 22 characters are not actually held in a location corresponding to any FORTRAN variable.

Alternatively, it may simply be required to print out a message from the program without actually inputting it, e.g.

```
      WRITE(6,20)
20 FORMAT(15HbENDbOFbPROGRAM)
```

would output the caption END OF PROGRAM when the WRITE statement was executed. Note that the number *w* preceding the H must take account of all the characters following the H, including blanks.

An alternative method of transmitting textual (or literal) information in some versions of FORTRAN is to place the character string between

quotes (this is not permitted in ASA FORTRAN). The following statements are equivalent

```
30 FORMAT('bTHEbROOTSbARE')
30 FORMAT(14HbTHEbROOTSbARE)
```

Similarly, if there are *w* blanks between the quotes, then *w* characters will be input when the FORMAT statement is used with a READ statement. If the character string itself contains an apostrophe, then this must be represented by two successive apostrophes if the whole string is enclosed in quotes. Thus the following are equivalent

```
25 FORMAT('bTHEbEQUATION''SbROOTSbARE')
25 FORMAT(25HbTHEbEQUATION'SbROOTSbARE)
```

A character string placed between quotes is generally referred to as literal data.

The X format specification which is of the form

*w*X

is used for transmitting blanks only. It does not require a corresponding FORTRAN variable in the I/O list, and in this sense it is similar to the H format specification. On input it has the effect of skipping *w* columns. It is useful for spacing numbers on a card without enclosing the blanks in a numeric field specification. It can also be used for skipping over comments in data. On output *w* blanks are inserted in an output record. This is useful for arranging results clearly on a line, e.g.

```
WRITE(6,1)X,Y,Z
1 FORMAT(F9.4,3X,F8.5, 4X,F8.3)
```

would produce a line of data arranged as follows

−589.7438*bbbb*0.05319*bbbbb*870.319

5.4.3 *Carriage control characters*

The lineprinter interprets the first character in every FORTRAN record as a carriage-control character. This character is not printed, but is used to determine the paper motion (if any) to precede the printing of the line. The four most frequently used carriage-control characters are:

b select a new line before printing.
0 select two new lines before printing.
1 go to the top of the next sheet of paper before printing.
+ print this line on top of the previous one (i.e. no paper motion).

If the carriage control character is any other character, then, in general, a fault will occur. It is important to realize that if the output is being directed to the card punch or any device other than the line-printer, then control characters are not required. If carriage-control characters are sent to the card punch, then they will be punched in column one.

The most common method of transmitting carriage-control characters is by means of the Hollerith or literal specifications, e.g.

```
FORMAT(1H1,F12.6,I3,F10.5)
FORMAT(12H0END bOF bRUNb,I3)
FORMAT('+',70X,'SPECTRUM',I4)
```

Note that the number of characters that can be printed on any one line is limited by the line width of the lineprinter concerned and is typically of the order of 120 characters.

The blank carriage-control character can also be transmitted using the X specification or by allowing one more space in a numeric field than is required to transmit the number, e.g. FORMAT(1X,F8.5) and FORMAT(F9.5) produce the same effect provided the field width required to print the number does not exceed 8.

Carriage-control characters are required at the beginning of every record to be printed; this also applies to new records introduced by means of a slash. Several slashes can be placed in succession. If n consecutive slashes appear at the beginning or end of a FORMAT statement, then on input n blank cards are skipped or on output n blank lines are inserted. If n consecutive slashes appear anywhere else in a FORMAT statement, then $n-1$ blank records are skipped. Unless the slashes are at the end of the statement, the last one must be followed by a carriage-control character for printed output.

5.4.4 *Double precision and logical fields*

The double precision and logical facilities in FORTRAN have not yet been discussed, and no attempt will be made to describe them here. A full treatment is given in Chapter 7. However, for completeness the format specifications used for inputting or outputting logical or double-length quantities will be stated here.

The format specification for transmitting double precision data is of the form

D$w.d$

where w is the field width and d the number of digits after the decimal point. This specification is entirely analogous to the E format code, and

the same rules apply (see Section 5.4.1). The exponent is specified by the letter D instead of E. Thus a number input on this format specification would be of the form

$$\pm\ 0.n_1n_2\ldots n_dD{+}m_1m_2$$

The format specification L_w is used for conversion between an internal logical value (which can only be .TRUE. or .FALSE.) and the external representation T or F; w here represents the total number of character positions occupied by the logical field. On input the first non-blank character must be either a T or an F, and will cause either .TRUE. or .FALSE. to be assigned to the corresponding logical variable in the I/O list. All characters following this in the field will be ignored. If the w characters are all blank a value .FALSE. will be assumed. On output a T or F is placed at the extreme right of the allotted space and is preceded by $w-1$ blanks.

5.5 Repeated fields

It may sometimes be found that the I/O list of a READ or WRITE statement consists wholly (or partly) of variables of the same type which have the same format specification. For example,

```
      READ(5,100)A,B,C
100   FORMAT(F8.4,F8.4,F8.4)
```

or

```
      READ(5,20)I,J,A,B
20    FORMAT(I5,I5,F8.4,F8.4)
```

To save the programmer from having to punch the same specification many times FORTRAN provides a means of denoting repeated fields of this type. Any unsigned integer constant placed before the format code denotes the number of times that specification is to be repeated. If there is no constant, then that specification is used once only. The above examples can be written

```
      READ(5,100)A,B,C
100   FORMAT(3F8.4)
```

and

```
      READ(5,20)I,J,A,B
20    FORMAT(2I5,2F8.4)
```

A group of format specifications can also be repeated by placing the group in parentheses and preceding it with an unsigned integer constant, e.g.

```
      READ(5,50)I,A,J,B
   50 FORMAT(2(I5,F8.4))
```

is equivalent to

```
      READ(5,50)I,A,J,B
   50 FORMAT(I5,F8.4,I5,F8.4)
```

Group specifications can be nested, i.e. a group can appear within another group.

The group format specification also affects the order in which the format codes are used. Earlier in this chapter (Section 5.2) the situation in which there are more variables in the I/O list than format specifications in the FORMAT statement was discussed. It was stated then that the FORMAT statement is repeatedly scanned from the beginning until the I/O list is exhausted. This is a perfectly valid statement unless the FORMAT statement contains group format specifications. In this case the FORMAT statement is rescanned from the last nested group (taking into account any repetition of that group). This is illustrated by means of the following examples:

```
      READ(5,20)I,J,A,K,B,L,C,M
   20 FORMAT(I5,2(I3,F10.6))
```

is equivalent to inputting data to I with format I5, J with I3, A with F10.6, K with I3 and B with F10.6. The FORMAT statement is then exhausted, a new card is selected and the last nested group specification (i.e.2(I3,F10.6)) is rescanned. L is input with format I3, C with F10.6 and M with I3. The I/O list is now exhausted, and so the rest of the FORMAT statement is ignored.

```
      READ(5,10)I,J,A,K,L,B,M,N
   10 FORMAT(I5,(I2,F9.3),I6)
```

In this situation I is input with format I5, J with I2, A with F9.3 and K with I6. The FORMAT statement is now exhausted, a new card is selected and the FORMAT statement is rescanned from the group (I2,F9.3). So L is input with format I2, B with F9.3 and M with I6. The FORMAT statement is again exhausted, a new card is taken and the statement is rescanned from the group (I2,F9.3). N is input with format I2, and the input list is now exhausted.

The general rule which applies is that the FORMAT statement is always rescanned from the rightmost left parenthesis (always taking into account any repetitions attached to the group within the parentheses).

Slashes may appear within group specifications to select new records.

5.6 Scale factors

A scale factor, P, can be used to change the position of a decimal point of a real number when it is transferred between the computer and the external medium.

On input it can only be used with the F format specification, i.e. it is of the form

*n*PF*w.d*

where *n* is an integer constant and *w* and *d* are as defined in Section 5.4. If *n* is positive the decimal point is moved *n* positions to the left. If it is negative the decimal point is moved *n* positions to the right. The external number 336.295 would be held internally as 3.36295 if input with a format specification 2PF8.3 and as 33629.5 if input with a format specification −2PF8.3.

On output a scale factor can be used with F, E or D specifications. The effect with the F format is exactly the same as for input, but the decimal point is moved in the opposite direction. With the E or D specification the value of the exponent is changed to correct for the movement of the decimal point, so that internally and externally the numerical value is identical.

A repetition number can still be placed in front of the F, E or D format code when a scale factor is used, i.e. 2P3F10.4 is perfectly valid and is equivalent to writing 2PF10.4, 2PF10.4, 2PF10.4.

Once a scale factor has been set, it remains set and applies to all following format specifications in the same FORMAT statement. The effect of a previous scale factor can be removed by using the 0P scale factor when required. For example, 2PF10.4, 0PF10.4 would input the data −2098.5632 and +1382.9685 in the form −20.985632, +1382.9685.

5.7 Run-time format statements

It may not always be possible or convenient for a programmer to specify the format of his data or results at the time of writing the program. For this reason FORTRAN provides a means of using variable FORMAT statements which can be read in as part of the data at run time and used by READ or WRITE statements later in the program. The format is read in as textual data with an A specification to any FORTRAN array. It must consist of a list of format specifications enclosed in parentheses but without the word FORMAT. Thus a data card could consist of the characters

(I5,*b*2(F10.4,3X),*b*E16.6)

Assuming 8 characters per word can be stored in the machine, then the above specification will occupy three words (including the commas,

brackets and blanks). The following statements will input this as part of the data and use it with subsequent READ and WRITE statements:

```
      DIMENSION FMT(3)
      READ(5,4)(FMT(I),I=1,3)
    4 FORMAT(3A8)
      .....
      .....
      READ(5,FMT)I,A,B,X
      .....
      .....
      WRITE(6,FMT)I,A,B,X
```

Note that the format is input as characters to an array FMT which must have been previously dimensioned (even if the array size is only 1).

If the format statement read in contains literal data (enclosed in quotes) which contains an apostrophe (i.e. a double quote), then it must only be used with an output statement. If it is required to use it for input, then the H format code must be used.

EXERCISES 5

(1) Write I/O statements together with associated FORMAT statements to:

(*a*) Input the data on card A to the variables N and M and the array I(6), and from N cards (of which a typical one is B) to the arrays X(N,3) and Y(N). The card B, for example, could be input to X(1,1), X(1,2), X(1,3) and Y(1).

CARD A:

bb5bb2b5b3b2b1b8b7

CARD B:

b—3.875*b*—8.7160*bbb*—0.571—6.8

(b) Output the arrays X(N,3) and Y(N) (as input above) in the form of a block as shown below:

	X	Y		X	Y		X	Y
	—3.87			—5.68			—6.51	
1	—8.72	4.0	2	—2.71	7.0	3	—0.79	1.0
	—0.57			—0.65			—0.99	
	—5.11			—6.71			—4.83	
4	—4.71	3.0	5	—3.87	2.1	6	—7.99	3.0
	—6.58			—2.98			—1.06	

etc.

(2) Write a program to plot a graph of the cosine function from 0 to 2π in 0.05π increments using the lineprinter. Assume that the lineprinter prints 120 characters in a line and that the variables AST and BL contain the character constants * and *b* which can be printed under A1 format specification. Try to improve the appearance of the graph by printing axes and by marking the 0, $\pi/2$, π, $3\pi/2$ and 2π axes and the 0, ± 0.5 and ± 1 ordinates.

CHAPTER SIX

Functions and subroutines

6.1 Introduction

Throughout this book an attempt has been made to point out that many problems can be coded into a FORTRAN program using only a small number of basic instructions. However, further facilities have been added to the FORTRAN language to simplify the programmer's task. The DO statement, for example, enables the programmer to perform a given computation many times without having to write it down more than once. In this chapter it will be seen that other facilities are available so that whole sections of a program need be written only once but can be referred to many times.

6.2 The arithmetic statement function

It may sometimes be found that the same arithmetic expression is calculated at several different points in the same program, although it may not necessarily involve the same FORTRAN variables on each occasion. In such circumstances it is possible to use an *arithmetic statement function* to define the expression once, attach a name to it and refer to it by this name throughout the rest of the program.

An arithmetic statement function definition is of the general form

NAME(*arg*)=EXPRESSION

where EXPRESSION is any valid arithmetic or logical expression which does not contain subscripted variables, NAME is the statement function name and *arg* is a list of *dummy arguments* separated by commas. The rules applying to NAME are the same as those for any ordinary FORTRAN variable name, and the first letter determines whether it is a real or integer function.

To illustrate the meaning of the term 'dummy arguments' an example of an arithmetic statement function will be considered. Suppose it is required to calculate the function

$$ax^2 + by^2 + cxy + d$$

at several points in a program. An arithmetic statement function named QUAD could then be used to define this function, i.e.

```
QUAD(X,Y)=A*X**2+B*Y**2+C*X*Y+D
```

This is simply a definition. No actual calculation is performed at this stage. It is therefore a non-executable statement, and it must appear before the first executable statement of the program. If it is required to calculate the value of this function at, say, three distinct points in the program for (X1,Y1), (X2,Y2) and (X3,Y3), then it is only necessary to write QUAD(X1,Y1), QUAD(X2,Y2) and QUAD(X3,Y3) at the relevant points, e.g.

```
RESULT=X4+QUAD(X1,Y1)-QUAD(X3,Y3)
```

X and Y in the function definition are only dummy arguments. No actual calculation will be performed with them; they will be replaced (when the function is activated) by the *actual arguments* X1, Y1, etc. X and Y as used in this definition will be quite distinct from any X or Y used in other parts of the program. Actual arguments and dummy arguments must correspond in order, number and type. An actual argument can be a constant, a subscripted or non-subscripted variable, an arithmetic or logical expression, an array name or the name of another subprogram (see the next section).

The expression on the right-hand side of an arithmetic statement function definition may also contain variables which do not appear in the argument list, e.g. A, B, C and D in the above example. In this case the A, B, C and D will be the *same* A, B, C and D which appear in the rest of the program, and their values must be assigned before the function is activated.

The expression may also contain other functions which must either be standard system functions (e.g., SIN, COS, etc., discussed in Section 2.2) or must have been defined earlier in the program.

6.3 The FUNCTION subprogram

The arithmetic statement function is labour-saving in that it enables the programmer to define once an expression which appears many times in a program. However, it is a very limited facility, since it can consist of one statement only and it produces one result. FORTRAN, therefore, provides an extension of this facility by allowing the programmer to name a *sequence* of statements as a function. The resulting FUNCTION *subprogram* is defined by a statement of the general form

FUNCTION *name*(*arg*)

where *name* is any valid FORTRAN name by which the subprogram will be identified throughout the rest of the program and *arg* is a list of dummy arguments separated by commas. The first letter of the name dentifies the function as being of type real or integer (this can be over-

ridden by preceding the word FUNCTION with one of the type declarations discussed in the next chapter). The dummy arguments must be non-subscripted variable or array names or the dummy names of other FUNCTION (or SUBROUTINE) subprograms.

The FUNCTION statement must be the first statement of the subprogram; it can be followed by any sequence of valid FORTRAN statements excluding another FUNCTION statement or a SUBROUTINE or BLOCK DATA statement (see later in this chapter). The physical end of the subprogram must be indicated by the END statement.

Thus a FUNCTION subprogram to calculate either the function $100(x_2 - x_1^2)^2 + (1 - x_1)^2$ or the function $(x_1 - x_2^2)^2 + (1 - x_2)^2$ according as the value of NOFUN is 1 or 2 could be written as follows:

```
      FUNCTION FX(X1,X2,NOFUN)
      GO TO (1,2),NOFUN
    1 F1=X2-X1*X1
      F2=1.0-X1
      FX=100.0*F1*F1+F2*F2
      RETURN
    2 F1=X1-X2*X2
      F2=1.0-X2
      FX=F1*F1+F2*F2
      RETURN
      END
```

The role of the RETURN statement will be discussed shortly.

The name of the FUNCTION subprogram must always appear at least once in the body of the subprogram. Generally it is placed on the left-hand side of an assignment statement (as in the example above) and the final required function value is assigned to it. It can also be placed in an input list within the subprogram or as an argument of a CALL statement (see later). It must *not* be referred to on the right-hand side of an arithmetic assignment statement.

The sequence of statements following a FUNCTION statement is referred to as a subprogram, since it is treated as an entirely separate program in compilation. This has two advantages. A large FORTRAN program is difficult to correct and test. By breaking it down into subprograms it can be written and tested in manageable sections; only when all separate parts are working satisfactorily will the entire program be run. Since a subprogram is a self-contained unit, it can be removed from any one program and run with any other program without any alteration. The FORTRAN variables used within the subprogram

have no relation to those used elsewhere in the program. Thus a variable X in one subprogram refers to a different location from a variable X in another.

All programs broken down into one or more subprograms in this way must also contain a *main program.* This is of the same form as any simple FORTRAN program as described in the earlier chapters of this book, except that it will contain references to other subprograms.

Although subprograms are independent units from the point of view of compilation, they must provide some means by which they can communicate with each other and with the main program during execution. One means of doing this is by the use of arguments (see the previous section). A FUNCTION subprogram can be activated by writing its name (followed by a list of actual arguments separated by commas) at the point in an expression where the value of the function is required. This causes control to be transferred to the subprogram, and the following statements are executed, replacing the dummy arguments (of which there must be at least one) by the actual arguments specified. The FUNCTION subprogram can be called from the main program or any other subprogram. When the execution of the subprogram is complete, control must be transferred back to the point in the calling program at which the function value is required. It is the RETURN statement which indicates the logical end of the subprogram and which causes control to be returned to the correct point. Several RETURN statements may be used to indicate the ends of different logical paths in the program, but there must always be at least one. The function FX defined earlier in this section can be activated as follows:

```
.....
DENOM=X*X-SQRT(FX(X,Y(4),1))
N1=2
.....
.....
RESULT=(1.0-FX(X+Y(1),Y(N1),N1))**2
.....
```

In the above example the actual arguments corresponding to the dummy variables X1, X2 (of type real) are the real variable X and the element Y(4) of a real array in the first call of FX and the real expression X+Y(1) and the array element Y(N1) in the second call. The integer dummy variable NOFUN is replaced by an integer constant 1 on the first call and an integer variable N1 on the second call. Note that the actual arguments always correspond in number (i.e. there are always three actual arguments), order (i.e. the two real quantities always precede the integer quantity) and type (i.e. a real quantity is substituted for a real dummy argument).

6.4 The SUBROUTINE subprogram

The FUNCTION subprogram described in the previous section had one important restriction: it had to return one, and only one, result. This restriction is removed by the SUBROUTINE subprogram, which can return no result (as in subroutine ERROR below) or many results to the calling program and is defined by a statement of the general form

SUBROUTINE *name*(*arg*)

where *name* is any valid FORTRAN name by which the subprogram will be identified and *arg* is a list of dummy arguments separated by commas. The important distinctions between this and a FUNCTION definition are, firstly, that no particular value is ever assigned to the name, so that the first letter has no particular significance, and it is meaningless for the name to appear anywhere in the body of the subprogram; secondly, the dummy argument list is optional – there need be no arguments at all. The dummy arguments must be non-subscripted variable or array names or the dummy name of another SUBROUTINE or FUNCTION subprogram.

The SUBROUTINE statement must be the first statement in the subprogram and can be followed by any sequence of valid FORTRAN statements, excluding the FUNCTION, SUBROUTINE or BLOCK DATA (see later) statements. The physical end of the subprogram must be indicated by the END statement, and it must contain either a RETURN statement or a STOP statement.

Values from the subprogram are returned to the calling program by means of one or more arguments in the argument list. Any variable used to transmit results in this way must be assigned a value within the subprogram by placing it on the left-hand side of an assignment statement, in an input list, as an argument in a function reference or as an argument in a CALL statement (to be discussed shortly).

Two typical subroutines are:

```
SUBROUTINE ROOTS(A,B,C,X1,X2)
B24AC=B*B—4.0*A*C
IF(B24AC.LT.0)CALL ERROR
B24AC=SQRT(B24AC)
X1=(—B+B24AC)/(2.0*A)
X2=(—B—B24AC)/(2.0*A)
RETURN
END
```

```
      SUBROUTINE ERROR
      WRITE(6,1)
    1 FORMAT('1COMPLEX ROOTS'/'PROGRAM TERMINATED')
      STOP
      END
```

Note that in these examples the dummy arguments A, B, C are used to supply information to the subroutine, whereas the dummy arguments X1, X2 are used to return the results to the calling program. ERROR is an example of a subprogram which does not require any dummy arguments and which does not contain any RETURN statement since it returns no results.

Since the name of a SUBROUTINE subprogram is merely a way of labelling that piece of program and has no particular numerical value associated with it, it is meaningless to write it as an operand in an expression. This form of subprogram is therefore activated by a CALL statement which is of the general form

CALL *name*(*arg*)

where *name* is the name of the SUBROUTINE to be entered and *arg* is a list of actual arguments (if any) which are to replace the dummy arguments. This causes the subroutine to be executed and control is returned (by the RETURN statement) to the statement immediately following the CALL statement in the calling program.

If it is required to calculate a function value (FX) and its derivative (G) at some point X, then a subroutine could be written of the form

```
SUBROUTINE CALCFG(X,FX,G)
.....
(coding to calculate the function and its derivative)
.....
RETURN
END
```

This subroutine could be entered at two different points in the main program by writing

```
.....
.....
CALL CALCFG(X1,FX1,G1)
.....
CALL CALCFG(X2,FX2,G2)
```

This would calculate the function and its derivative at the points X1 and X2, returning the results in FX1,G1 and FX2,G2.

6.5 The EXTERNAL statement

If a subprogram requires the name of another subprogram as one of its arguments, then this name must be declared in an EXTERNAL statement in the calling program. This declaration is of the general form

EXTERNAL *name1*, *name2*, . . .

where *name1*, *name2*, etc., are the names of the subprograms that are to be passed as arguments to other subprograms. The statement must precede any function definition statements.

The EXTERNAL statement distinguishes the subprogram name from that of a FORTRAN variable name. An EXTERNAL statement must be used even if the subprogram concerned is a standard system function, e.g.

Calling Program	*Subprogram*
EXTERNAL SIN,SQRT	SUBROUTINE SUBR(X,F,Y)
CALL SUBR(2.0,SIN,RESULT)	Y=F(X)
WRITE(6,1)RESULT	RETURN
1 FORMAT(F10.5)	END
CALL SUBR(2.0,SQRT,RESULT)	
WRITE(6,1)RESULT	
STOP	
END	

would cause first the sine and then the square root of 2.0 to be computed. In each case the value is returned in RESULT.

6.6 Adjustable dimensions

If it is required to use subscripted variables within a subprogram, then the arrays concerned must be dimensioned within the subprogram. It was stated earlier (in Section 6.3) that one of the advantages of subprograms was that they enabled one generalized section of coding to be used with many different programs without any alteration. However, a subprogram may contain an array A with dimension 100, but one program may require an array with 50 elements, while another may require 200 elements. Since this would require some alteration to the

subprogram, it would be more convenient if the dimensions of an array could be adjusted from outside the subprogram. Thus, statements of the form

```
DIMENSION A(N),B(M,N)
```

are allowed in subprograms. N and M are integer variables specifying the dimensions of the arrays within the subprogram. These must be assigned values in the calling program, and N, M and the array names must be passed to the subprogram in the argument list. The arrays concerned must be dimensioned absolutely in the calling program, and the values of N and M must not exceed this absolute value.

An array in COMMON (see next section) must not be used with adjustable dimensions.

6.7 The COMMON and EQUIVALENCE statements

Normally subprograms are completely self-contained units; a variable X in one subprogram refers to a different location in the computer from a variable X in another subprogram. However, it may be that the programmer intends the X to refer to the same location in both cases. This could be ensured by placing X in the argument list. An alternative method (and frequently a more convenient one, particularly if many variables are involved) would be to declare X a *common variable* by writing

```
COMMON X
```

The effect of this statement is to reserve a location (named X) in a special area of store which is accessible by all subprograms (normally the store is divided up so that an area reserved for a particular subprogram is accessible by that subprogram only). Thus if X is a COMMON variable, then a reference to X in any subprogram refers to that one location in the COMMON store, provided X has been declared as COMMON in that subprogram.

The COMMON statement is a non-executable statement and must be placed before any reference to a COMMON variable. A whole string of variables can be declared COMMON in one COMMON statement, e.g.

```
COMMON X,Y,Z
```

reserves the first location in COMMON store for X, the second for Y and the third for Z. Whole arrays can also be declared COMMON. They may be dimensioned either in the COMMON statement, e.g.

```
COMMON A(10)
```

or in a DIMENSION statement preceding the COMMON statement, e.g.

```
DIMENSION A(10)
COMMON A
```

In the former case the array A must *not* also appear in a DIMENSION statement. Both forms have the effect of reserving ten locations in COMMON store for the array A.

The COMMON statement can also be used as a method of saving storage space. If the statement

```
COMMON X,Y,Z
```

appears in two subprograms then references to X,Y or Z in either subprogram will access the first, second or third location in COMMON store. However, if one subprogram contains

```
COMMON X,Y,Z
```

and the second contains

```
COMMON P,Q,R
```

then the first statement causes the first three locations in COMMON to be allocated to X, Y and Z, and the second statement causes the *same* three locations to be allocated to P,Q,R. Thus this has the effect of equivalencing X and P, Y and Q, Z and R, i.e. two (or more) distinct FORTRAN variable names in different subprograms can refer to the same location.

In the examples given so far only *blank common* store has been described, i.e. no name has been used to label a particular block of COMMON store: the entire COMMON area has been referred to. However, it is possible to place variables and arrays in separate COMMON areas by giving each area any valid FORTRAN name through which it can be identified. This is referred to as labelled (or named) common. The name must be placed in slashes and be followed by a list of the variables (separated by commas) which are to be assigned to that COMMON block, e.g.

```
COMMON A,B,C/AREA1/X,Y,Z/AREA2/P,Q,R/COUNTS/C1,C2
```

causes A,B,C to be reserved in blank COMMON, X,Y,Z to be reserved in the COMMON block named AREA1, P,Q,R in the COMMON block named AREA2 and C1,C2 in the COMMON block named COUNTS.

This facility is useful in that it enables a subprogram to share a COMMON block with one subprogram and a different COMMON block with another subprogram.

It was stated earlier that the COMMON statement can be used as a

method of equivalencing FORTRAN variables in different subprograms. A statement is also provided which allows variables to be equivalenced in the same subprogram without them necessarily referring to COMMON storage locations. The statement is of the general form:

EQUIVALENCE (*list1*),(*list2*)

where *list1* and *list2* represent lists of subscripted or non-subscripted FORTRAN variables (separated by commas) which are to be equivalenced, e.g.

EQUIVALENCE (X,Y),(A,B,C)

causes X and Y to refer to the same location and A,B,C to refer to the same location.

Whole arrays can be equivalenced simply by declaring the first elements in an EQUIVALENCE statement, e.g.

```
DIMENSION A(50),B(50),C(50)
EQUIVALENCE (A(1),B(1),C(1))
```

will equivalence all 50 elements of the arrays A, B, and C.

Two variables in one COMMON block or in two different COMMON blocks cannot be equivalenced.

6.8 The DATA statement and the BLOCK DATA subprogram

The programmer may occasionally find it more convenient to assign some data to his program variables at compile time rather than to read it all in at run time. This can be carried out by means of the DATA statement, which is of the general form:

DATA *list1*/*k1***d1*,*k2***d2*, . . . *kn***dn*/,*list2*/*k1***d1*,*k2***d2*,
. . . *km***dm*/ . . .

where *list1*, *list2*, etc., represent lists of variables, subscripted variable or array names; *d1*, *d2*, . . . *dn* (*dm*) represent integer, real, literal, logical or complex data constants which are to be assigned to the corresponding element in the preceding list; *k1*, *k2*, . . . *kn*(*km*) are optional and represent unsigned integer constants giving the number of consecutive variables to be assigned the particular constant. For example,

DATA A,B,C/67.87,54.72,5.0/

assigns the values 67.87 to A, 54.72 to B and 5.0 to C.

DATA DOT,X,BLANK/1H.,1HX,1H /

assigns the characters '.' to DOT, 'X' to X and '*b*' to BLANK and

```
DIMENSION A(20),Z(5)
DATA A,Z,J/20*0.0,5*1.5,3/
```

assigns the twenty elements of the array A the value zero, the five elements of the array Z the value 1.5 and the integer variable J the value 3.

Data can also be assigned to variables in labelled COMMON, but this must be carried out by means of a BLOCK DATA subprogram which must not contain any executable statements. It is of the general form

```
BLOCK DATA
.............
.............
END
```

The subprogram can include only DATA, COMMON, DIMENSION, EQUIVALENCE and Type statements (see next chapter). The COMMON statement must precede the DATA statement.

CHAPTER SEVEN

Further aspects of FORTRAN

7.1 Type statements

Any FORTRAN variable has an implicit type associated with the first letter of its name. A variable beginning with I, J, K, L, M or N is taken to be an integer variable, and a variable beginning with any other letter is taken to be real. This system, however, has two disadvantages. Firstly, the programmer may wish to call some real variable by a name which begins with one of the integer letters. For example, it is customary in physics or engineering to denote a current by the letter *I* and inductance by the letter *L*. The programmer may wish to keep these letters for his own convenience, but FORTRAN variables I and L have an implied integer type, whereas current and inductance are real quantities. Secondly, the implicit type declarations are restricted to two types of variable (integer and real), whereas FORTRAN provides six types. There must be some means therefore of declaring the type of variable in such a way that it overrides the implicit type.

This can be done by any of the following type statements:

INTEGER,REAL,DOUBLE PRECISION,LOGICAL,
EXTERNAL,COMPLEX

Any of these statements followed by a list of any number of variables separated by commas declares those variables to be of that type regardless of the initial letter of the name. Thus

REAL I,L

means that throughout the following program I and L will be taken to be real variables. The type statement must appear before any other reference to the variables concerned. Note that double precision, logical and complex variables must always be declared as such, since otherwise they would automatically be taken to be single precision real or integer variables. Note also that the EXTERNAL statement (see Section 6.5) is a type statement, since it defines a name to be a subprogram name rather than a simple variable name.

The type statements are non-executable statements.

7.2 Double precision constants and variables

In long numerical computations appreciable rounding errors can be introduced, since numbers can only be held in the computer to a

certain precision limited by the word length of the particular machine. The accumulation of rounding errors can be quite significant when the computation involves the addition or subtraction of very large or very small numbers. Therefore it is necessary to provide programmers with the facility to declare any variable to be double length so that two words, instead of one, are assigned to this name and a number can be held to at least twice the precision (and on some machines a little more than twice). Similarly, constants can be held to double precision.

A constant is taken to be double precision if, when written in floating-point form, the letter D is used to denote the exponent instead of the letter E. Thus 1.5D0 represents the number 1.5 held to double precision, and 1.62D4 represents the number 16200 held to double precision. This also applies to constants written in an expression in a program and to double precision numbers read in as data. The rules which apply to the form of the constants are exactly the same as for ordinary single precision floating-point constants written with an E exponent (see Section 1.3).

FORTRAN variables are declared to be double precision by the type statement DOUBLE PRECISION, e.g.

```
DOUBLE PRECISION DIFF,PRODCT,SUM
```

Double precision expressions use the same operators (+ − * / **) as single precision expressions and the same rules of precedence apply. Double precision quantities can be mixed with real or integer quantities in an expression: the integer will first be converted to a double-length floating point form. The mixed expression will yield a double-length result. If this is assigned to a single-length variable the most significant part will be taken. If it is assigned to an integer variable, then it will be truncated.

Data can be input or output to double length by means of the format specification D*w.d*, which is similar to the E format specification discussed earlier (see Section 5.4.1).

Functions can be declared double precision by preceding the word FUNCTION with the type specification DOUBLE PRECISION. Most installations provide standard double-length system functions. Some common ones are:

DSQRT	finds the square root of a double-length argument
DSIN	finds the sine of a double-length argument
DCOS	finds the cosine of a double-length argument
DATAN	finds the arctangent of a double-length argument
DEXP	finds the exponential of a double-length argument
DLOG	finds the natural logarithm of a double-length argument
DABS	finds the absolute value of a double-length argument

All the functions yield a double-length result.

7.3 Complex constants and variables

A complex number is represented in the computer as a pair of real numbers, the first being the real part and the second the imaginary part. A complex constant consists of two real numbers (which can be written in any allowable fixed or floating-point form) separated by commas and enclosed in parentheses. Thus the constant (5.6,3.2) represents the complex number $5.6+3.2i$.

A complex variable must be declared to be complex in the type statement COMPLEX, which must appear before any reference to the variable concerned. The initial letter of the variable name then has no particular significance. Two words will be allocated to the complex variable and their contents will be two real quantities.

Arithmetic can be performed with complex variables using the operators + − * /. These have the following effect. Consider two complex variables A and B where

$$A=a+bi$$

$$B=c+di$$

Then

$$A+B=(a+bi)+(c+di)=(a+c)+(b+d)i$$

$$A-B=(a+bi)-(c+di)=(a-c)+(b-d)i$$

$$A*B=(a+bi)*(c+di)=(ac-bd)+(ad+bc)i$$

$$A/B=(a+bi)/(c+di)=\frac{ac+bd}{c^2+d^2}+\frac{(bc-ad)i}{c^2+d^2}$$

Complex quantities can be mixed with integer, real or double-length quantities in expressions. Before an operation is performed between two operands the integer, real or double-length quantity is converted to a complex form. The expression yields a complex result. If the result of a complex expression is assigned to a real or double-length variable, then the real part only is assigned; the imaginary part is not used. If a complex quantity is assigned to an integer variable, then the real part is truncated and assigned.

The operator ** can be used only to raise a complex quantity to an integer power.

Complex quantities are input or output as two separate real quantities using an F or E format specification. (For example, READ (5,1) C(I,J), where FORMAT statement 1 is of the form FORMAT (2F 10.5)).

Functions can be declared complex by preceding the word FUNC-

TION with the type specification COMPLEX. Some common complex functions provided by the system are:

CSQRT	finds the square root of a complex argument
CCOS	finds the cosine of a complex argument
CSIN	finds the sine of a complex argument
CEXP	finds the exponential of a complex argument
CLOG	finds the natural logarithm of a complex argument
CABS	finds $(a^2+b^2)^{\frac{1}{2}}$ for the complex argument $a+bi$

7.4 Logical constants and variables

In Section 3.6 logical expressions were introduced, and it was described then how a logical expression could be constructed from relational expressions and the logical operators

.AND. .OR. .NOT.

In addition, logical constants and variables can be combined by these operators. Logical constants and variables can only have two values – either true or false. The logical constant .TRUE. represents the value true and .FALSE. represents the value false. Logical variables must be declared in the type specification LOGICAL, and can be assigned the values .TRUE. or .FALSE. by means of an assignment statement of the general form

LOGICAL VARIABLE=LOGICAL EXPRESSION

where LOGICAL EXPRESSION can be a logical constant or variable, a relational expression or any combination of these using the logical operators, e.g.

```
L1=.TRUE.
L2=L3
L4=A.GT.25.0.OR.L5
```

are valid assignments (L1, L2, L3, L4 and L5 are logical variables).

Logical quantities can be input or output by means of the L format specification which has been described earlier (see Section 5.4.4).

APPENDIX ONE

A least-squares curve-fitting program

Figures I (i) and (ii) show a complete listing of a FORTRAN program to find the coefficients of any polynomial (of any specified degree less than 10) which best fits a given set of data. The Figures consist of listings of two Subroutines (NORMEQ and GAUSS) and a main program which was used to test the Subroutines on an IBM 360/65 computer.

A brief description of the mathematical method now follows. The theoretical curve which best fits any given set of experimental data is found by minimizing the sum of the squares of the deviations at each point. If at some point x_i the experimental curve has the value y_i^{exp} and the theoretical curve the value y_i^{th}, then the smaller the expression

$$\sum_1^N (y_i^{exp} - y_i^{th})^2$$

the better the fit. Therefore to find the second-degree polynomial

$$y_i^{th} = c_1 x_i^2 + c_2 x_i + c_3$$

which best fits a given set of data the coefficients c_1, c_2 and c_3 which minimize the expression

$$\sum_1^N (y_i^{exp} - c_1 x_i^2 - c_2 x_i - c_3)^2$$

must be found.

This is achieved by differentiating the above expression with respect to c_1, c_2 and c_3, equating the resulting expressions to zero and solving the three simultaneous equations for c_1, c_2 and c_3. These three equations (which are referred to as the *normal equations*) are of the form:

$$Nc_3 + \left(\sum_1^N x_i\right) c_2 + \left(\sum_1^N x_i^2\right) c_1 = \sum_1^N y_i^{exp}$$

$$\left(\sum_1^N x_i\right) c_3 + \left(\sum_1^N x_i^2\right) c_2 + \left(\sum_1^N x_i^3\right) c_1 = \sum_1^N (x_i y_i^{exp})$$

$$\left(\sum_1^N x_i^2\right) c_3 + \left(\sum_1^N x_i^3\right) c_2 + \left(\sum_1^N x_i^4\right) c_1 = \sum_1^N (x_i^2 y_i^{exp})$$

where N is the total number of data points.

```
      C MAIN PROGRAM FOR LEAST SQUARES CURVE FITTING
      C NOTE MAXIMUM NUMBER OF DATA POINTS IS 100
      C AND MAXIMUM DEGREE IS 10
    1       DIMENSION X(100),Y(100),COEF(11)
    2       INTEGER DEGREE,DEGP1
      C READ DATA
    3       READ(5,1)DEGREE,NOPTS,(X(I),Y(I),I = 1,NOPTS)
    4     1 FORMAT(I2,I5/(2F10.4))
    5       DEGP1 = DEGREE + 1
    6       CALL NORMEQ(DEGREE,DEGP1,NOPTS,X,Y,COEF)
      C PRINT RESULTS
    7       WRITE(6,3)(I,COEF(I),I = 1,DEGP1)
    8     3 FORMAT('1ORDER          COEFFICIENT'/(1X,I5,2X,F15.7))
    9       STOP
   10       END
   11       SUBROUTINE NORMEQ(DEGREE,DEGP1,NOPTS,X,Y,COEF)
      C THIS SUBROUTINE SETS UP THE NORMAL EQUATIONS
   12       INTEGER DEGT2,DEGREE,DEGP1
   13       DIMENSION POWX(200),X(NOPTS),Y(NOPTS),SUM(11,11),
           1RHS(11),COEF(DEGP1)
   14       DEGT2 = DEGREE*2
      C COMPUTES SUMS OF POWERS
   15       DO 1 I = 1,DEGT2
   16       POWX(I) = 0.0
   17       DO 1 J = 1,NOPTS
   18     1 POWX(I) = POWX(I) + X(J)**I
      C FROM THE SUMS OF POWERS COMPUTE THE LHS OF THE
      C EQUATIONS IN THE TWO DIMENSIONAL ARRAY 'SUM'
   19       DO 2 I = 1,DEGP1
   20       DO 2 J = 1,DEGP1
   21       K = I + J - 2
   22       IF(K.LE.0)GOTO 3
   23       SUM(I,J) = POWX(K)
   24       GOTO 2
   25     3 SUM(I,J) = NOPTS
   26     2 CONTINUE
      C SET UP THE RHS OF THE EQUATIONS
   27       RHS(1) = 0.0
   28       DO 4 J = 1,NOPTS
   29     4 RHS(1) = RHS(1) + Y(J)
   30       DO 5 I = 2,DEGP1
   31       RHS(I) = 0.0
   32       DO 5 J = 1,NOPTS
   33     5 RHS(I) = RHS(I) + Y(J)*X(J)**(I - 1)
      C CALL SUBROUTINE GAUSS TO SOLVE THE EQUATIONS
   34       CALL GAUSS(DEGREE,DEGP1,RHS,COEF,SUM)
   35       RETURN
   36       END
```

Figure I (i)

```
37          SUBROUTINE GAUSS(DEGREE,DEGP1,RHS,COEF,SUM)
    C THIS SUBROUTINE SOLVES THE SET OF EQUATIONS
    C SUM(I,J) = RHS(I)
38          INTEGER DEGREE,DEGP1
39          DIMENSION RHS(DEGP1),COEF(DEGP1),SUM(11, 11)
40          DO 10 K = 1,DEGREE
    C CARRY OUT THE ELIMINATION PROCESS 'DEGREE' TIMES
41          KPLUS1 = K + 1
42          L = K
    C FIND TERMS OF GREATEST MAGNITUDE
43          DO 11 I = KPLUS1,DEGP1
44          IF(ABS(SUM(I,K)).LE.ABS(SUM(L,K)))GOTO 11
45          L = I
46        11  CONTINUE
    C IF TERMS ARE SUITABLY ORDERED THEN OMIT INTERCHANGE
47          IF(L.LE.K)GOTO 12
    C INTERCHANGE ROWS TO OBTAIN TERMS OF DECREASING
    C MAGNITUDES
48          DO 13 J = K,DEGP1
49          DUMP = SUM(K,J)
50          SUM(K,J) = SUM(L,J)
51        13  SUM(L,J) = DUMP
52          DUMP = RHS(K)
53          RHS(K) = RHS(L)
54          RHS(L) = DUMP
55        12  DO 10 I = KPLUS1,DEGP1
    C FIND FACTOR WHICH WILL ELIMINATE TERM AFTER
    C SUBTRACTION
56          FACTOR = SUM(I,K)/SUM(K,K)
    C SET THIS TERM = 0
57          SUM(I,K) = 0.0
58          DO 14 J = KPLUS1,DEGP1
    C COMPUTE OTHER TERMS
59        14  SUM(I,J) = SUM(I,J) - FACTOR*SUM(K,J)
60        10  RHS(I) = RHS(I) - FACTOR*RHS(K)
    C COMPUTE SOLUTION OF SINGLE EQUATION REMAINING
61          COEF(DEGP1) = RHS(DEGP1)/SUM(DEGP1,DEGP1)
62          I = DEGREE
63        16  IPLUS1 = I + 1
64          TOTAL = 0.0
    C COMPUTES OTHER SOLUTIONS BY SUBSTITUTION
65          DO 15 J = IPLUS1,DEGP1
66        15  TOTAL = TOTAL + SUM(I,J)*COEF(J)
    C RETURN SOLUTIONS IN THE VECTOR 'COEF'
67          COEF(I) = (RHS(I) - TOTAL)/SUM(I,I)
68          I = I - 1
69          IF(I.GT.0)GOTO 16
70          RETURN
71          END
```

Figure I (ii)

This theory can be extended to fit a polynomial of degree M. The normal equations which must be solved are now of the form:

$$N c_{M+1} + \left(\sum_1^N x_i\right) c_M + \ldots \left(\sum_1^N x_i{}^M\right) c_1 = \sum_1^N y_i{}^{exp}$$

$$\left(\sum_1^N x_i\right) c_{M+1} + \left(\sum_1^N x_i{}^2\right) c_M + \ldots \left(\sum_1^N x_i{}^{M+1}\right) c_1 = \sum_1^N (x_i y_i{}^{exp})$$

$$\left(\sum_1^N x_i{}^M\right) c_{M+1} + \left(\sum_1^N x_i{}^{M+1}\right) c_M + \ldots \left(\sum_1^N x_i{}^{2M}\right) c_1 = \sum_1^N (x_i{}^M y_i{}^{exp})$$

The necessary summations must be performed and the equations solved for the coefficients c_1 to c_{M+1}.

```
b7bbbb8
b0.0bbbbbb1.0bbbb
b0.25bbbbb0.9689b
b0.5bbbbbb0.8776b
b0.75bbbbb0.7317b
b1.0bbbbbb0.5403b
b1.25bbbbb0.3153b
b1.5bbbbbb0.0707b
b2.0bbbbbb-0.4161
```

(*a*) Typical data cards

ORDER	COEFFICIENT
1	0.9999521
2	0.0040209
3	−0.5230428
4	0.0483149
5	−0.0046293
6	0.0203557
7	−0.0046212

(*b*) Typical results

Figure I (iii) Typical Input and Output for the Least Squares Curve Fitting Program.

Subroutine NORMEQ performs the required summations and sets up the normal equations which are solved by the Gaussian Elimination Method (see, for example, R. W. Hamming, *Numerical Methods for Scientists and Engineers*, p. 360, McGraw-Hill, 1962) in Subroutine GAUSS.

The coefficients of a polynomial of degree DEGREE (restricted to a maximum of 10 in this case) which best fits a number, NOPTS, of data points X,Y (less than or equal to 100) are calculated. The values of the coefficients are returned in the array COEF. DEGP1 is set to DEGREE+1 in the main program before the Subroutine is called. The sums, which are the coefficients in the normal equations, are accumulated in the array SUM and passed from NORMEQ to GAUSS via the argument list. Similarly, the constant terms on the right-hand side of the normal equations are accumulated in the array RHS. The array POWX holds the powers of X from 1 to 2*DEGREE.

Since the program has a straightforward logical flow, a flow chart is not shown here. Figure I (iii*a*) shows the form of a typical Input to the program and Figure I (iii*b*) a typical Output.

APPENDIX TWO

A Kutta–Merson numerical integration program

Figures II (i) and (ii) show a complete listing of a FORTRAN program to advance, by one step, the integration of a system of differential equations of the general form:

$$\left(\frac{dy}{dx}\right)_i = f_i(x, y_1, y_2, \ldots y_n)$$

The integration is performed using the Kutta–Merson Method (see L. Fox, *Numerical Solution of Ordinary and Partial Differential Equations*, p. 24, Pergamon Press, 1962).

A listing of a typical main program to test the Subroutine KUTTAM which performs the Integration is shown in Figure II (i), together with the subroutine DYBYDX, which must be written to supply the values of $\left(\frac{dy}{dx}\right)_i$ at the required positions.

```
      C MAIN PROGRAM TO TEST SUBROUTINE KUTTAM
 1          DIMENSION Y(10),SOLN(10)
 2          REAL INSTEP
 3          LOGICAL START
 4          START = .TRUE.
 5          READ(5,1)X,INSTEP,ACC,NEQS,(Y(I),I = 1,NEQS)
 6      1   FORMAT(3F8.3,I3/10F8.3)
 7          CALL KUTTAM(NEQS,X,Y,ACC,INSTEP,START,SOLN)
 8          WRITE(6,2)(SOLN(I),I = 1, NEQS)
 9      2   FORMAT(18H1VALUES OF Y ARE ,10F10.5)
10          STOP
11          END


64          SUBROUTINE DYBYDX(X,Y,DYDX,NEQS)
      C THIS SUBROUTINE DEFINES THE DIFFERENTIAL EQUATIONS
65          DIMENSION Y(NEQS),DYDX(NEQS)
66          DO 1 I = 1,NEQS
67      1   DYDX(I) = - 2.0*X*Y(I)*Y(I)
68          RETURN
69          END
```

Figure II (i)

```
12          SUBROUTINE KUTTAM(NEQS,X,Y,ACC,INSTEP,START,SOLN)
13          DIMENSION Y(NEQS),Y0(10),Y1(10),Y2(10),DYDX0(10),
           1DYDX1(10),DYDX2(10),SOLN(NEQS)
14          LOGICAL START,DOUBLE
15          REAL INSTEP
16          IF(.NOT.START)GOTO 1
    C IF NOT FIRST ENTRY THEN SKIP INITIALISATION INSTRUCTIONS
17          DO 2 I=1,NEQS
18        2 Y0(I)=Y(I)
    C INITIALISE COUNTERS
19          STEP=INSTEP
20          NSREQD=1
21          START=.FALSE.
22          GOTO 3
23        1 STEP=INSTEP/NSREQD
24        3 NSDONE=0
    C COMPUTE 5 APPROXIMATIONS TO Y
25       10 CALL DYBYDX(X,Y0,DYDX0,NEQS)
26          DO 4 I=1,NEQS
27        4 Y1(I)=Y0(I)+STEP/3.0*DYDX0(I)
28          CALL DYBYDX(X+STEP/3.0,Y1,DYDX1,NEQS)
29          DO 5 I=1,NEQS
30        5 Y1(I)=Y0(I)+STEP/6.0*DYDX0(I)+STEP/6.0*DYDX1(I)
31          CALL DYBYDX(X+STEP/3.,Y1,DYDX1,NEQS)
32          DO 6 I=1,NEQS
33        6 Y1(I)=Y0(I)+STEP/8.0*DYDX0(I)+3.0*STEP/8.0*DYDX1(I)
34          CALL DYBYDX(X+STEP/2.0,Y1,DYDX2,NEQS)
35          DO 7 I=1,NEQS
36        7 Y1(I)=Y0(I)+STEP/2.0*DYDX0(I)-3.0*STEP/2.0*DYDX1(I)+
           12.0*STEP*DYDX2(I)
37          CALL DYBYDX(X+STEP,Y1,DYDX1,NEQS)
38          DO 8 I=1,NEQS
39        8 Y2(I)=Y0(I)+STEP/6.0*DYDX0(I)+2.0*STEP/3.0*DYDX2(I)+
           1STEP/6.0*DYDX1(I)
40          DOUBLE=.TRUE.
41          DO 9 I=1,NEQS
    C COMPUTE ERROR ON EACH Y VALUE
42          ERROR=ABS(0.2*(Y1(I)-Y2(I)))
43          IF(ERROR.LE.ACC)GOTO 11
    C IF ACCURACY IS NOT REACHED THEN HALVE STEP LENGTH
44          STEP=STEP/2.0
45          NSREQD=2*NSREQD
46          NSDONE=2*NSDONE
47          GOTO 10
48       11 IF(ERROR*64.0.GT.ACC)DOUBLE=.FALSE.
49        9 CONTINUE
    C PREPARE FOR ANOTHER STEP
50          X=X+STEP
51          DO12 I=1,NEQS
52       12 Y0(I)=Y2(I)
53          NSDONE=NSDONE+1
    C IF REQUIRED STEP COMPLETED THEN GOTO 13
54          IF(NSDONE.GE.NSREQD)GOTO 13
55          IF(.NOT.(DOUBLE.AND.NSDONE.EQ.(NSDONE/2)*2.AND
           1NSREQD.GT.1))GOTO 10
    C IF ACCURACY TOO GREAT DOUBLE THE STEP LENGTH
56          STEP=2.0*STEP
57          NSDONE=NSDONE/2
58          NSREQD=NSREQD/2
59          GOTO 10
    C PUT FINAL VALUES OF Y IN ARRAY SOLN
60       13 DO 14 I=1,NEQS
61       14 SOLN(I)=Y0(I)
62          RETURN
63          END
```

Figure II (ii)

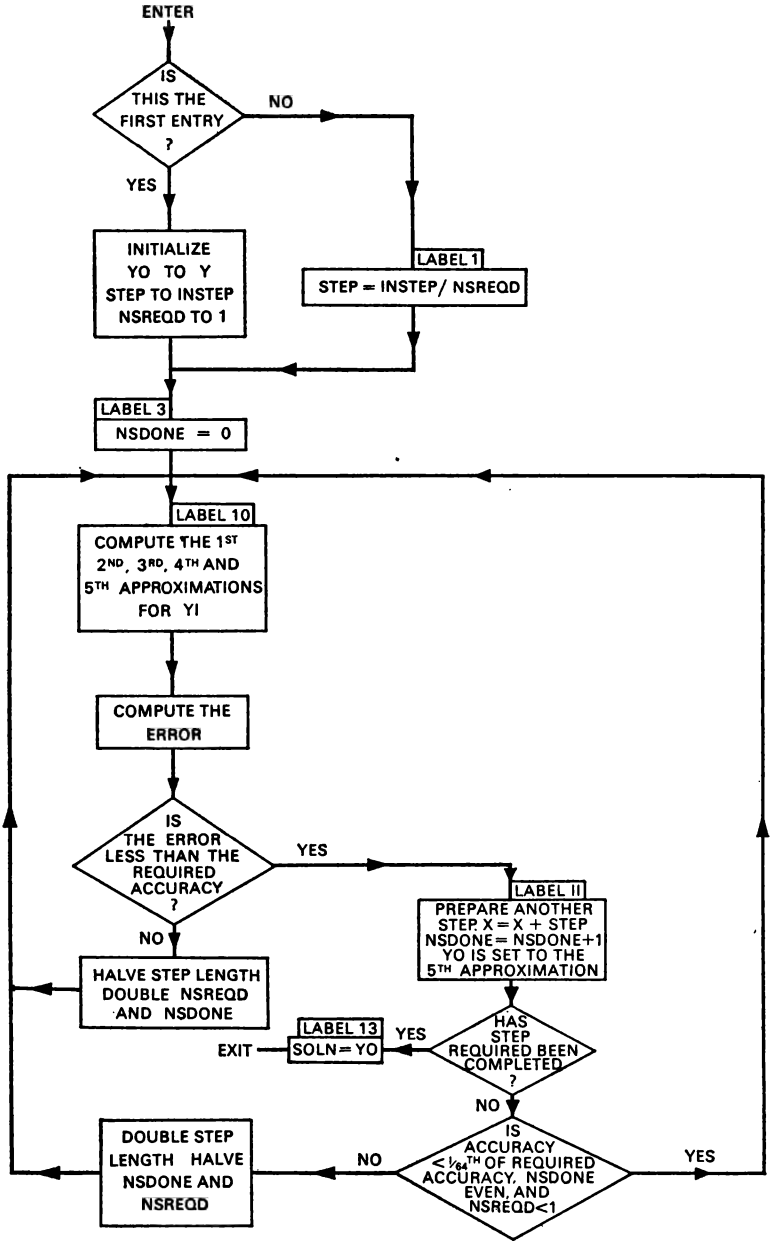


Figure II (iii) Flow chart of Subroutine KUTTAM

The main program reads in the current x position (X), the accuracy required (ACC), the number of differential equations involved (NEQS), the step across which the integration is to be performed (INSTEP) and the corresponding values of y at the point x (Y).

In the sample subroutines shown in Figure II (i) the Subroutine KUTTAM is called to integrate the differential equation

$$\left(\frac{dy}{dx}\right) = -2xy^2$$

In KUTTAM the differential equations are expanded to fifth order, and an estimate of the accuracy of the method is obtained by taking a fifth of the difference between the fourth and fifth terms. In each iteration this is tested against the required accuracy; if this has not been reached the step length is decreased and more steps are taken to cover the interval. NSREQD and NSDONE are used to keep track of the number of steps required to complete the interval and the number of steps completed so far.

KUTTAM is normally entered with the logical variable START equal to .TRUE.. In this case the procedure is initialized by setting Y equal to Y0, STEP equal to INSTEP and NSREQD=1. If the subroutine is entered with the value of START equal to .FALSE., then the initialization is omitted and STEP is set equal to INSTEP/NSREQD. In this way an integration which has been performed to a given accuracy can be performed again to a greater accuracy. On exit from KUTTAM, START always has the value .FALSE..

The required values of y are returned in the array SOLN.

A flow diagram of the subroutine is shown in Figure II (iii) and typical Input and Output in Figures II (iv*a*) and II (iv*b*).

*b*1.0*bbbbb*1.0*bbbb*0.000001*bb*1
*b*0.5*bbbb*

(*a*) Typical data cards

VALUES OF Y ARE 0.20000

(*b*) Typical results

Figure II (iv) Typical input and output for the KUTTA–MERSON Numerical Integration Program.

In the example given, an accuracy of 1 in 10^6 was reached in less than $\frac{1}{3}$second on the IBM 360/65 computer.

Solutions to exercises

Exercises 1

(1) The following are correct integer constants:

2225 −25 −97612 10000 0 005326

The following are correct real constants:

.137 1.562 −.137 +0162.2E−02 +258. 3.6E−22

The following are not valid:

5.2E100 (more than two digits in the exponent)

2E−7 (no decimal point)

E−10 (exponent alone not allowed; this must be written 1.E−10)

101.2E+2. (exponent not an integer)

2,360 (comma not allowed)

(2) The following are correct integer variable names:

NUMBER I123 ITER IP L1369P

The following are correct real variable names:

PI AMPS FRED ZETA Q COUNTS POWER

The following are not correct:

2X13 (does not begin with a letter)

M63−2 (contains a character other than a letter or a digit)

BSQUARED (contains more than six characters)

X+Y (contains a character other than a letter or a digit)

K(2) (contains a character other than a letter or a digit)

INC* (contains a character other than a letter or a digit)

NUMBER1 (contains more than six characters)

L3.6 (contains a character other than a letter or a digit)

JIM'S (contains a character other than a letter or a digit)

Exercises 2

(1) (*i*) A=(3.*X*Y**2*(Z+1.)+0.5*Y*Z)/(1.+X+X**3)
(*ii*) B=1.5*X*(X−1.)*SQRT(7.*Y−LOG(COS(X)))
(*iii*) I=K/J*(X*1.E+2−K*K/(3.*J))
(*iv*) Z=5.*X*X*SQRT((COS(X*X−Y*Y))**3+ATAN
(X*COS(X)))/(EXP(X+1.)*EXP(Y+1.)+1.)

In case (*iii*) when J=3, K=7 and X=0.5, I=89

```
(2)     READ(5,1)NPOUND,NSHILL,NPENCE,RATE,TERM
      1 FORMAT(I4,I3,I3,F10.5,F10.5)
        SUM=NPENCE+12*(NSHILL+20*NPOUND)
        YIELD=SUM*(1.+RATE/100.)**TERM−SUM
        NPOUND=YIELD/240.
        DOLLRS=YIELD/100.
        YIELD=YIELD−NPOUND*240.
        NSHILL=YIELD/12.
        NPENCE=YIELD−NSHILL*12.+0.5
        WRITE(6,2)NPOUND,NSHILL,NPENCE,DOLLRS
      2 FORMAT(I5,I3,I3,F10.2)
        STOP
        END
```

Exercises 3

(1) (*a*) true
(*b*) false
(*c*) true

(2) J=7

(3) The following program accumulates the sum of the first N integers in the FORTRAN variable ISUM:

```
      READ(5,1)N
    1 FORMAT(I4)
      ISUM=0
C NOTE THAT AN INTEGER SUM IS ACCUMULATED
      I=0
    3 IF(I.GT.N)GOTO 2
C IF I IS NOT GREATER THAN N COMPUTE THE NEXT
C SQUARE AND ADD IT IN TO ISUM
      ISUM=I*I+ISUM
      I=I+1
C REPEAT THE PROCESS
      GOTO 3
C WHEN THE PROCESS IS COMPLETE PRINT THE
C ANSWER
    2 WRITE(6,4)ISUM
    4 FORMAT(I7)
      STOP
      END
```

(4) The following is a typical program to evaluate the square root (B) of a number (X):

```
      READ(5,1)X
    1 FORMAT(F10.5)
      A=X/2
    2 B=(X/A+A)/2
      C=B-A
      IF(C.LT.0)C=-C
      IF(C.LT.10.E-6)GOTO 3
      A=B
      GOTO 2
    3 WRITE(6,1)B
      STOP
      END
```

Exercises 4

(1) The errors in the program given are as follows:

(*a*) There is no DIMENSION statement before line 1. This should be of the form

```
DIMENSION NO(500),X(500)
```

This would limit the program to a maximum of N=500.

(*b*) NSMODD is not initialized to zero before line 6. To correct for this insert

```
5a NSMODD=0
```

(*c*) The DO loop at lines 13 and 14 contains two mistakes. Firstly, a DO statement cannot contain an expression (i.e. N−1). Secondly, in line 14 a subscript of 0 will be generated for NO(I−1) when I=1. A zero subscript is not allowed in FORTRAN, and so to correct for this the first element of X must be computed separately outside the loop as is the last element. To correct for both these mistakes line 13 must be replaced by:

```
13a X(1)=NO(2)
13b NF=N−1
13c DO 7 I=2,NF
```

(*d*) The integer division at line 14 will give a truncated result which in this case is incorrect. This can be corrected for by dividing by the real number 2.0.

(*e*) The value of I is undefined on exit from the loop at line 15, and here it is assumed that it is equal to the final value N. Line 15 should therefore be written:

```
15 X(N)=NO(N−1)
```

(*f*) At line 17 X(I) is output with an integer format specification. This should be replaced by

```
17a   WRITE(6,9)X(I)
17b 9 FORMAT(F10.5)
```

(2)
```
    DIMENSION N1(7,5),N2(7,5),N3(7,5),N4(5,7)
    DO 1 I=1,5
    DO 1 J=1,7
    N3(J,I)=N1(J,I)+N2(J,I)
1   N4(I,J)=N3(J,I)
```

(3) The following program reads N numbers into the array X, calculates the largest (XMAX), the smallest (XMIN), the mean (XMEAN) and the standard deviation (STDEV):

```
      DIMENSION X(1000)
      READ(5,1)N
    1 FORMAT(I4)
      IF(N.GT.1000)STOP
      SUM=0.
      SQDIFF=0.0
      XMIN=10.E10
      XMAX=0.
      DO 2 I=1,N
      READ(5,3)X(I)
    3 FORMAT(F10.3)
      SUM=SUM+X(I)
      IF(X(I).GT.XMAX)XMAX=X(I)
      IF(X(I).LT.XMIN)XMIN=X(I)
    2 CONTINUE
      XMEAN=SUM/N
      DO 4 I=1,N
    4 SQDIFF=(X(I)-XMEAN)**2+SQDIFF
      STDEV=SQRT(SQDIFF/N)
      WRITE(6,5)XMAX,XMIN,XMEAN,STDEV
    5 FORMAT(F10.4,F8.4,F9.4,F10.5)
      STOP
      END
```

Exercises 5

(1) (*a*)
```
      READ(5,1)N,M,I,((X(J,I),I=1,3),Y(J),J=1,N)
    1 FORMAT(2I3,6I2/(3F8.3,F4.1))
```

(*b*)
```
      WRITE(6,2)
      DO 4 J=1,N,3
      K=J+2
    4 WRITE(6,3)(X(L,1),L=J,K),(L,X(L,2),Y(L),L=J,K),
     1 (X(L,3),L=J,K)
    2 FORMAT(1H1/1H0,3(5X,1HX,4X,1HY,2X))
    3 FORMAT(/1H ,3(3X,F5.2,5X)/1H ,3(I2,1X,F5.2,F4.1,1X)
     1/1H ,3(3X,F5.2,5X))
```

(2)
```
      DIMENSION CH(120)
      J=10
      WRITE(6,1)
    1 FORMAT('1'/' COSINE GRAPH '//10X,'−1',22X,'−0.5',
     123X,'O',23X,'0.5',23X,'1'/11X,'|',3(24X,'|'))
      DO 2 I=1,41
      X=(I−1)*0.05
      IF(J.EQ.10)GO TO 3
      J=J+1
      WRITE(6,4)
    4 FORMAT(61X,'|')
      GOTO 5
    3 J=1
      WRITE(6,6)X
    6 FORMAT(11X,'+',2(24X,'+'),F4.2,20X,'+',24X,'+')
    5 DO 7 M=1,120
    7 CH(M)=BL
      NOSP=61.5+50.*COS(X)
      CH(NOSP)=AST
    2 WRITE(6,8)CH
    8 FORMAT('+',120A1)
      STOP
      END
```

Index

Science Paperbacks 65

A Course on Programming in FORTRAN IV
V. J. Calderbank

This book provides for beginners a short, readable course on programming in FORTRAN IV, which is at present one of the most widely used scientific computer languages. The text assumes only a very elementary knowledge of mathematics but the appendices contain examples of complete FORTRAN IV programs which may be of practical use to more advanced programmers. The description of the language develops progressively throughout the book, new facilities being introduced only when a sufficient number of examples and exercises have been provided to ensure competence in more basic aspects of the language. The reader is encouraged to write complete FORTRAN IV programs at an early stage. Although the book describes one particular version of FORTRAN IV which is in widespread use (the IBM 360 FORTRAN IV), care is taken to point out the differences from ASA FORTRAN IV.

In the same series

Computer Operating Systems
D. W. Barron

The Theory of Computer Science
A Programming Approach
J. M. Brady

The BASIC Idea
An Introduction to Computer Programming
R. Forsyth

Computers
G. M. Phillips and P. J. Taylor

Scientists Must Write
A guide to better writing for scientists, engineers and students
R. Barrass

A complete list of Science Paperbacks is available from t

aperbacks are p
and Hall
Fetter Lane Lonc